UNIVERSITETET I OSLO
Institutt for informatikk

MPRP: Parallellisering På Multiprosessor

Masteroppgave

Kristoffer Skaret

6. mai 2008

Sammendrag

Datamaskinen har siden dens barndom hatt en kraftig ytelsesøkning. I de senere årene har man begynt å ta i bruk parallellisering for å opprettholde denne effekten. Parallell programmering er vanskelig, og det er derfor utviklet verktøy som forenkler jobben. JavaPRP er en preprosessor som automatisk genererer parallelle programmer beregnet på nettverk av maskiner. Denne oppgaven bringer ideene fra JavaPRP over på multiprosessor. En multiprosessor er en datamaskin med mange prosessorer som er knyttet tett sammen. Den største fordelen til parallelle programmer for multiprosessor er at de kan nyttiggjøre felles hukommelse.

Denne oppgaven presenterer ulike parallelle varianter av tre typer algoritmer. De parallelle programmene utnytter multiprosessorens felles hukommelse på ulike måter. Det blir så drøftet hvordan man kan foreta automatisk parallellisering på multiprosessoren. Løsningen er et nytt system der det er hentet inspirasjon fra både JavaPRP og de tre ulike parallelle programmene. Preprosessoren MPRP tilbyr en enkel og fleksibel behandling av tre typer sekvensielle algoritmer slik at multiprosessorens egenskaper utnyttes godt.

Forord

Denne masteroppgaven avslutter mitt studium ved Institutt for Informatikk ved Universitetet i Oslo. Dette har vært en lang og innholdsrik opplevelse, og det er flere personer jeg vil takke.

Først vil jeg takke min veileder Arne Maus som tilbød meg denne spennende oppgaven. Arne har bidratt med mange gode tips samtidig som han har hjulpet meg til å styre oppgaven i en interessant retning. Arne har også vært en viktig inspirasjonskilde når jeg har gått tom for kreativitet.

Så vil jeg takke hele min familie som har vært tilstede og støttet meg under den lange studietiden.

Jeg vil også takke mine gode venner som jeg har blitt kjent med under studietiden. Dere bidratt med masse hjelp og tips, og ikke minst gjort studiene til en morsom og minneverdig opplevelse. Spesielt vil jeg takke Lars som har lest gjennom hele oppgaven og kommet med gode forslag.

Til slutt vil jeg rette en stor takk til min kjære Linda som har hjulpet meg masse under hele oppgaveskrivingen. Linda har gitt støtte og inspirasjon når det har stått på som verst, og hun har korrekturlest alt jeg har skrevet og hjulpet meg med gode forslag og tips.

Insitutt for Informatikk
Blindern, Mai 2008

Kristoffer Skaret

Innhold

Kapittel 1

Innledning

De mange anvendelsesområdene gjør datamaskinen til et hjelpemiddel i de aller fleste sammenhenger. Et viktig bruksområde i dag er at datamaskiner nyttes til å løse ulike typer matematiske og beregningstunge problemer. Siden dens barndom har vi dessuten kunnet nyte godt av en eksponentiell utvikling av datamaskinenes yteevne, og et sterkt og vedvarende prisfall. Man har fått behov for å behandle større datamengder, det blir benyttet større programmer, og det har dukket opp mer ressurskrevende problemer man ønsker å løse. De stadig kraftigere maskinene gjør det mulig å realisere disse målene.

Vi ser likevel at det fortsatt er behov for raskere maskiner. Man kan tenke seg mange problemer som dagens maskiner er for svake til å løse, men som morgendagens datamaskiner trolig vil være i stand til å håndtere. Et eksempel kan være at det innefor meteorologi trolig i fremtiden vil være mulig å benytte simuleringsmodeller så omfattende at man vil kunne spå været over en klart lengre tidshorisont enn det som brukes i dag.

Det er egentlig bare fantasien som setter grenser for hvilke problemer man kan tenke seg å løse ved hjelp av datamaskiner. Noen problemer er så store at man trolig aldri vil klare å løse dem, selv ikke om en hadde hele universets plass og levetid til rådighet.

Målet er derfor fortsatt å lage datamaskiner med bedre yteevne, og det finnes mange fremgangsmåter for å oppnå dette. En måte er å lage raskere maskinvare som bruker kortere tid på beregningene eller som kan håndtere mer data. En annen måte er å optimalisere programvare og algoritmer. En tredje teknikk er parallellisering. Analyseselskapet Gartner la nylig frem det de tror vil være de syv viktigste utfordringene innenfor IT i år 2033. En av disse syv er parallell programmering [Gar08].

Parallellisering vil si å la flere datamaskiner samarbeide om å løse ett problem.

1.1 PRP - Parallelle rekursive Prosedyrer

Det å bygge opp et parallelt system fra bunnen av kan være en vanskelig og tidkrevende prosess. Det ideelle ville derfor være om man hadde et program som kunne hjelpe brukeren å parallellisere programmer. Denne tanken ble først presentert av Arne Maus i 1978 [MA95]. Siden den gang har det gradvis blitt utarbeidet et system som tar seg av nettopp dette. Siste versjonen av systemet heter JavaPRP. Systemet benytter maskiner som er tilknyttet internett for å parallellisere programmer skrevet i Java.

1.2 Multiprosessor

I den senere tid har det dukket opp en ny type parallelle maskiner. Multiprosessoren er en parallell maskin med flere prosessorer knyttet tett sammen. Der PRP hittil har dreiet seg kun om parallellisering over nettverk vil denne oppgaven gå et skritt videre og ta for seg automatisk parallellisering på multiprosessor. Denne oppgaven vil derfor ta for seg ulike måter å parallellisere på multiprosessor, og hvilke utfordringer disse byr på. Den vil også handle om de fordeler multiprosessoren gir og hvordan man best kan utnytte disse under parallelliseringen.

Basert på kunnskapen om multiprosessoren er målet å finne ut hvordan man på en god måte kan utføre automatisk parallellisering på denne. Det vil bli designet et nytt system, MPRP (Multicore PRP), som skal ta seg av dette. MPRP vil i likhet med JavaPRP være et system som på en brukervennlig måte kan generere parallelle programmer basert på sekvensielle programmer. MPRP skal best mulig utnytte fordelene og håndtere utfordringene man har ved parallellisering på multiprosessoren.

1.3 Oppbygning av oppgaven

Her følger en kort beskrivelse av hva de ulike kapitlene i denne oppgaven inneholder.

Kapittel 2, **Datamaskinens utvikling**, beskriver hvordan datamaskinene har blitt stadig raskere, og forklarer hvorfor man i de senere årene har gått over til å benytte parallellitet for å opprettholde denne effekten. Kapittel 3, **Parallellitet**, handler om ulike typer parallelle systemer. Så gir kapittel 4, **JavaPRP**, en introduksjon til PRP-prosjektet som ligger til grunn for denne oppgaven. Kapittel 5, **Multiprosessor**, handler om hvordan multiprosessoren kan benyttes til parallell problemløsning. Det legges særlig vekt på hvordan felles hukommelse fungerer og kan nyttiggjøres i parallelle programmer.

Kapittel 6, 7 og 8 presenterer tre ulike parallelliserbare problemer. Problemene skiller seg fra hverandre med hensyn på hvordan deres algoritme

er oppbygd. Derfor må parallelliseringen av disse foregå på ulike måter, og det blir gitt forslag til ulike algoritmer og programmer som løser disse i parallell på multiprosessor.

Kapittel 9, **Automatisk parallellisering**, drøfter ulike måter å implementere et system for generering av parallelle programmer beregnet på multiprosessor. Det konkluderes med at den beste løsningen er et helt nytt system skreddersydd for parallellisering på multiprosessor. Kapittel 10, **Presentasjon av MPRP**, forklarer hvordan dette nye systemet virker. Systemet er inspirert av JavaPRP og parallelliseringen av de tre problemene fra de tidligere kapitlene.

Kapittel 11, **Oppsummering og videre arbeid**, inneholder konklusjon og noen forslag til hva som kan være aktuelt å jobbe videre med.

Til slutt følger to tillegg. Tillegg A, **Brukerveileding for MPRP**, forklarer hvordan systemet MPRP virker og viser et eksempel på bruk. Tillegget er skrevet på engelske for at det skal kunne være tilgjengelig for flere. Tillegg B inneholder kildekoden til tre ulike programmer, før og etter preprosessering av MPRP.

Kapittel 2

Datamaskinens utvikling

Gordon Moore observerte datamaskinens kraftige utvikling i 1965. Han så at transistorene, de små elektriske komponentene som en prosessor består av, lot seg produsere i en stadig mindre størrelse. Han forutså dermed at man ville oppnå en dobling av antall transistorer det er plass til på en brikke omtrent hver attende måned. En slik halvering av transistorstørrelsen innebærer grovt sett en dobling av ytelse, fordi det gjør at man kan benytte seg av mer avansert prosessorarkitektur og at man kan tillate seg høyere klokkefrekvens. Denne trenden har vist seg å la seg gjelde frem til i dag, og er nå kjent under navnet "Moores lov". [Moo05]

Navn	Introd.	Antall trans.	Tr.bredde	Kl.frekvens
Pentium	1993	3.2 millioner	$0.6\mu m$	100 MHz
Pentium II Desch.	1998	7.5 millioner	$0.25\mu m$	400 MHz
Pentium III Cop.	2000	28 millioner	$0.18\mu m$	1000 MHz
Pentium 4 Nor.	2002	55 millioner	$135nm$	3 GHz
Pentium M Dothan	2004	77 millioner	$90nm$	2.2 GHz
Core 2 Conroe	2006	291 millioner	$65nm$	3 GHz

Tabell 2.1: Ulike mikroprosessorer fra Intel [Wik08b]

Tabell 2.1 viser eksempler på noen av prosessorene som har blitt produsert av selskapet Intel de siste årene. Tabellen viser at transistorstørrelsen, eller transistorbredden, har krympet i raskt tempo. Det har gjort at klokkefrekvensen og antall transistorer det er plass til på brikkene har latt seg øke. Klokkefrekvensen beskriver hvor mange instruksjoner, eller enkeltberegninger, prosessoren utfører per tid og det er derfor ønskelig med så høy klokkefrekvens som mulig. Med andre ord viser tabellen at man omtrent hvert halvannet år har kunnet forvente å bytte ut en datamaskin med en som er dobbelt så rask. Utviklingen er altså i tråd med Moores lov fra 1965.

2.1 Varmeutvikling

En prosessor utvikler mye varme. Denne varmen må fraktes bort, for en overopphetet prosessor slutter å fungere. Varmeutviklingen er større jo høyere klokkefrekvens prosessoren benytter. Varmen setter imidlertid en øvre grense for hvor høy klokkefrekvens en prosessor kan takle.

Varmeutviklingen stammer opprinnelig fra transistorene som prosessoren er bygd opp av. En moderne prosessor kan bestå av flere hundre millioner transistorer. Deres oppgave er å styre strømføringen i prosessoren, og de virker som en brytere som skrur strømmen av eller på. Temperaturen øker fordi det forekommer et lite effekttap hver gang en av de mange transistorene skifter status mellom av og på. Dette effekttapet blir til varme. Som vist i tabell 2.1 på forrige side kan nyere prosessorer ha klokkefrekvens på opp til 3 GHz. Transistorene i en moderne prosessor kan altså skifte status flere milliarder ganger pr sekund, og til sammen vil alle de små effekttapene utgjøre en stor oppheting av prosessoren.

Problemet er at for høy temperatur gjør at en transistor ikke alltid virker som den skal. Varmen fører til at det lekker en liten lekkasjestrøm gjennom transistoren. Denne strømmen kan gjøre at en transistor som skal være skrudd av kan oppfattes som om den nesten er skrudd på. Lekkasjestrømmen blir større jo varmere prosessoren er.

Oppvarmingen av prosessorer motvirkes ved hjelp av kjøling med for eksempel vifter.

2.2 Stagnering

I alle år har den typiske måten for å øke ytelsen til prosessorene vært å gjøre transistorene mindre. Da oppnår man høyere klokkefrekvens, og man får plass til flere transistorer på samme brikke.

I dagens prosessorer finner man transistorer med en bredde helt ned i 65 nanometer, og de blir stadig mindre. Neste skritt er transistorbredde på 45 nm, og det er allerede planlagt produksjon av transistorer med bredde på 10 nm [Int07].

Problemet er at for mindre transistorer er lekkasjestrømmene mer betydelige enn for større transistorer. Det kommer av at det er mindre motstand for lekkasjestrømmen i en liten transistor. Prosessorene krever derfor stadig mer kjøling for at man skal kunne oppnå ønsket klokkefrekvens. En moderne prosessor med klokkefrekvens på 3 GHz krever kjøling ved hjelp av for eksempel en kraftig vifte kombinert med kjøleribber. Hvis man samler mange slike prosessorer på et sted, for eksempel i en serverpark, kreves det dessuten ytterligere nedkjøling av selve rommet maskinene står i.

Slikt kjøleutstyr er kostbart, særlig med tanke på energiutgifter. Man har derfor de senere årene blitt nødt til å la varmeutviklingen i større grad

begrense hvor høy klokkefrekvens man benytter. Det er mye som tyder på at vi faktisk har kommet til et nivå hvor det ikke lenger er hensiktsmessig å øke frekvensen ytterligere. Av tabell 2.1 på side 5 kan man se at taket ser ut til å ligge på omtrent 3 GHz. Siden Gordon Moores tid og frem til omtrent år 2002 har man kunnet følge en jevn utvikling av prosessorenes klokkefrekvens. I 2002 nådde man 3 GHz, og helt frem til i dag, 6 år senere, har man ikke kunnet se noen videre utvikling på området. Dagens prosessorer benytter vanligvis en klokkefrekvens på mellom 2 og 3 GHz.

I supercomputeren BuleGene/L, avbildet på bilde 3.1 på side 9, har man faktisk valgt å senke klokkefrekvensen for å begrense effektforbruket. Den parallelle datamaskinen består av svært mange prosessorer som hver trolig er i stand til å kjøre med klokkefrekvens på opp til 3 GHz. Likevel har man valgt å la prosessorene kjøre på 800MHz og benytte et større antall prosessorer for å kompensere for ytelsestapet. Det gjør at effektforbruket senkes betraktelig.

Som vi har sett har vi fortsatt behov for å utvikle raskere datamaskiner. Men fordi varmeutviklingen ser ut til å sette en grense for hvor høy klokkefrekvens vi kan benytte er vi nå nødt til å ty til andre teknikker. En metode kan være å utvikle datamaskiner basert på helt annen teknologi som gir høyere ytelse uten like stor varmeutvikling. Blant annet har det blitt foreslått å kombinere dagens silisiumteknologi med såkalte nanorør for å kunne lage svært små transistorer [Kan03].

En annen metode er å lage datamaskiner som benytter seg av parallellisering. Denne teknikken er mer og mer brukt i dagens maskiner, og de parallelle datamaskinene er trolig i ferd med å ta over for de rent sekvensielle.

Kapittel 3

Parallellitet

Parallellisering dreier seg om å la flere prosessorer samarbeide om å løse det samme problemet. Det er nærliggende å tro at man kan løse et problem raskere dersom man deler problemet opp i mindre deler og lar flere prosessorer samarbeide om å løse det. Dette har vist seg å stemme for veldig mange problemer.

Parallellisering kan utføres på mange måter. Vi kan dele inn i ulike typer maskiner, og ulike typer programmer.

3.1 Parallelle datamaskiner

I dag finnes det mange forskjellige typer parallelle datamaskiner, og mange måter å klassifisere dem. Man kan gjøre fysiske klassifiseringer der man for eksempel skiller mellom hvordan og hvor tett prosessorene er koblet sammen, eller man kan skille mellom ulike former for organisering av minne.

Figur 3.1: BlueGene/L ble i november 2007 kåret til verdens raskeste supercomputer. Den parallelle datamaskinen er i stand til å utføre 478.2 billioner kalkulasjoner per sekund [TOP08]

.

Det gis her en beskrivelse av de vanligste klassifiseringene.

3.1.1 Løst og fast koblet system

Det er vanlig å skille mellom hvor tett prosessorene i systemet er knyttet sammen. I et fast koblet system er prosessorene tett knyttet sammen gjennom for eksempel samme databuss. Systemet er bygd opp slik at man kjenner nøyaktig hvilke komponenter det består av og hvordan de er koblet sammen. Prosessorene i systemet kan ha delt hukommelse.

Et løst koblet system består av en rekke enkeltstående datamaskiner som er koblet sammen ved hjelp av et kommunikasjonssystem som for eksempel Ethernet. Man kan dele inn i flere kategorier av løst koblede systemer. Dersom man bygger en parallell datamaskin ved å koble mange datamaskiner sammen i et lokalt nettverk får man en såkalt klynge. Maskinene er koblet sammen etter et egnet mønster, og man har full kontroll på de enkelte maskinenes oppbygning og ressurser. Fordelen med slike systemer er at kan oppnå like høy ytelse som en enkeltstående maskin, men til en mye lavere pris.

Den største ulempen med et løst koblet system er at det er lang fysisk avstand mellom prosessorene. Det gjør at kommunikasjonen mellom disse tar lang tid. Derfor egner løst koblede systemer seg best til å løse problemer der det er lite behov for kommunikasjon mellom prosessorene. Fordelen er at løst koblede systemer ofte er billigere å produsere enn fast koblede systemer.

3.1.2 Grid

En annen type løst koblet system er et såkalt grid [FKT01]. Hovedforskjellen mellom grid og klynger er at datamaskinene i et grid kan være plassert på ulike steder i verden.

Et grid er en virtuell supercomputer som egentlig består av mange enkeltstående datamaskiner. En supercomputer er en svært rask men også svært kostbar datamaskin som brukes til å løse store problemer. Gridsystemene kan brukes til å løse tilsvarende store problemer, men til en lavere pris. Det blir billigere fordi grid slår sammen datakraften til mange relativt rimelige enkeltsystemer rundt i verden.

Et gridsystem kan fungere ved at ulike organisasjonene deler sine datamaskinressurser med andre organisasjoner. Grid dreier seg da om organiseringen av akkurat dette. De ulike organisasjonene kan tilby ulike tjenester. Det kan være tilbud om alt fra raske beregninger til spesielle applikasjoner eller lagringsplass. Organisasjonene i nettverket kan dermed finne de ressursene de trenger på datamaskiner hos organisasjoner andre steder i verden.

Gridsystemer kan også benytte seg av det faktum at mange tradisjonelle PC-er sjelden utnytter all datakraften de har tilgjengelig. Tanken er at en PC som bare kjører enkle tekstbehandlere og lignende har mye overskudd av datakraft. Ved å dele dette overskuddet med andre kan maskinen bli en del av et grid. Et av de mest kjente slike systemene er SETI@home der over tre millioner brukere har bidratt med datakraft til å søke etter signaler fra utenomjordisk liv. I skrivende stund har prosjektet omlag 800000 aktive brukere som gir en total beregningskraft på 461 TeraFLOPS [BOI08]. Det gjør at systemet kan måle seg med verdens raskeste supercomputer som er avbildet på figur 3.1 på side 9.

3.1.3 Organisering av minne

Parallelle datamaskiner kan organisere minnet sitt på mange forskjellige måter. Vi kan dele inn i to hovedkategorier:

- **Distribuert minne**
  Dette betegner en datamaskin der hver prosessor har sitt eget lokale minne. Dette er typisk for løst koblede systemer

- **Delt minne**
  Her benytter de forskjellige prosessorene seg av samme minne. Kommunikasjon kan da foregå ved å lese og skrive til dette minnet.

Kombinasjoner av dette kan også forekomme, der prosessorene både har eget og delt minne.

3.1.4 Flynns modell

En annen måte å skille mellom ulike datamaskinarkitekturer på er en klassifisering gjort av Michael J. Flynn [Fos95]. Modellen deler maskiner inn i kategorier basert på antall samtidige instruksjoner og antall samtidige datastrømmer som forekommer. Hovedtypene er som følger:

- **SISD (Single Instruction Single Data)**
  En slik maskin utfører en instruksjon om gangen på et sett av data. Ingen parallellisering forekommer. Dette er den typiske sekvensielle maskinen vi er vant med å bruke.

- **SIMD (Single Instruction Multiple Data)**
  Dette er en parallell datamaskin som kan utføre samme sekvens av instruksjoner i parallell på mange ulike data. I sammenhenger det man har behov for å utføre de samme beregningene på mye data kan dette være nyttig, men det er dermed altså nødvendig at problemet som skal løses kan deles opp etter denne modellen. Disse maskinene kan for eksempel være gode på å løse matrisemultiplikasjon effektivt.

GPU (graphical processing unit) er eksempel på en slik datamaskin. Det er den prosessoren som sitter på grafikkortet i en PC, og som tar seg av å tegne grafikken som skal vises på skjermen.

- **MISD (Multiple Instruction Single Data)**
  Denne typen datamaskin kan utføre flere ulike instruksjoner på samme data. Her er det naturlig å tilordne de ulike instruksjonene til hver prosessor, og la hver enkelt av dem utføre instruksjonene de har blitt tildelt på dataene.

  Pipelinearkitektur tilhører denne typen. Det er noe som moderne CPU-er benytter seg av for å kunne parallellisere instruksjoner som opprinnelig er skrevet sekvensielt.

- **MIMD (Multiple Instruction Multiple Data)**
  På denne typen datamaskin får hver prosessor sitt eget sett med data og utfører sitt eget sett med instruksjoner. Eksempler på en slik maskin er et nettverk av maskiner eller en multiprosessor. Dette er den mest allsidige typen parallell datamaskin.

3.2 Parallelle programmer

Det er vanskeligere å programmere parallelle programmer enn å programmere sekvensielle programmer. Derfor er det utviklet ulike modeller og metoder for hvordan slike systemer kan designes. Ulike metoder kan egne seg til forskjellige problemer og maskiner. Man kan dele inn etter for eksempel instruksjons- og dataflyt, størrelse på datamengden som parallelliseres, eller teknikk for ressursallokering og kommunikasjon. Det finnes også systemer som tar seg av å automatisere parallelliseringen av et ellers sekvensielt program.

Dette avsnittet gir en oversikt over noen av disse systemene og modellene.

3.2.1 Meldingsutveksling

Har man et løst koblet system med distribuert minne kan programmene som kjører på systemet typisk kommunisere ved hjelp av meldingsutveksling. En slik melding kan for eksempel bestå av et funksjonskall eller en datapakke. Et system som er lagd for dette er MPI (Message Passing Interface) [Fos95].

MPI er et grensesnitt for meldingsutveksling. Ved bruk av MPI opprettes en eller flere prosesser som kommuniserer via dette grensesnittet. MPI tilbyr hele 129 funksjoner for å ta seg av sending om mottak av disse meldingene. Blant disse finner man globale operasjoner slik som for eksempel Barrier Sync, som sørger for synkronisering av prosesser.

3.2.2 Oppgaveinndeling

En annen vanlig fremgangsmåte for oppdeling er å lage en algoritme som består av en rekke individuelle oppgaver. Slik kan oppgavene fordeles mellom de ulike prosessorene og utføres i parallell. De ulike oppgavene må tilordnes en prosessor, og det kan gjøres på mange måter. Det er vanlig å ha flere oppgaver enn prosessorer og tilordne mange oppgaver til hver prosessor. På den måten tilbyr man skalerbarhet med hensyn på antall prosessorer. Oppgaver som er knyttet sammen ved at de for eksempel jobber på samme data bør tilordnes samme prosessorer eller prosessorer som er tett knyttet sammen. I praksis starter man med å opprette en oppgave som kjører på en prosessor, og den oppgaven tar seg så av å opprette nye oppgaver og tilordne disse til en annen CPU.

Tråder og maskiner

Parallelle systemer kan som tidligere nevnt enten benytte seg av mange maskiner, eller de kan kjøre på en enkelt maskin med flere CPU-er.

Ved bruk av mange maskiner er det vanlig å kjøre ett program på hver enkelt av de som kan kommunisere gjennom for eksempel meldingsutveksling eller oppgaveinndeling. Klynger og grid er eksempler på dette.

For systemer som skal kjøre på kun en maskin med mange CPU-er er det vanlig å lage et program med mange tråder. Tråder er en måte å strukturere programmer på. Et program som skal utføre flere ulike selvstendige oppgaver kan tilordne en tråd til hver av disse oppgavene. Trådene kan rangeres og dele på den prosessorkraften som er tilgjengelig. Har man tilgang til flere prosessorer kan trådene parallelliseres ved å utnytte de forskjellige prosessorene.

3.2.3 Ulik størrelsesorden av parallelle programmer

Parallellisering kan foregå på ulike nivåer med hensyn på størrelsen til den problembiten man ønsker å parallellisere. Noen systemer parallelliserer på et verdensomspennende omfang ved å benytte seg av mange maskiner koblet til internett, mens andre parallelliserer på kretsnivå på en enkelt brikke.

Tabell 3.1 viser noen slike nivåer i stigende rekkefølge.

Instruksjoner

Den minste formen for parallellisering kan man finne vi inne på selve CPU. Moderne CPU-er som utfører instruksjonene i et sekvensielt program er ofte i stand til å utføre flere av disse instruksjonene på samme tid. Dette er en automatisk prosess som foregår uten at programmereren behøver å

	Størrelsesnivå	Parallelliserer på
1	Instruksjonssett	Instruksjoner
2	Løkker	Programblokker
3	Rekursjon	Metoder
5	Prosesser	Programmer

Tabell 3.1: Ulike størrelsesnivåer av parallellitet. Tabellen viser forskjellige måter parallellisering kan foregå på i stigende rekkefølge.

tenke på parallellisering. Også andre typer prosessorer kan utføre slik miniatyrparallellisering. På GPU-en foregår det på en litt annen måte. Den består av et stort antall identiske beregningsenheter som samtidig utfører samme instruksjon på hver sin datadel.

Løkker

Et høyere nivå er å fokusere på løkker i et program. Et system som bygger på programmeringsspråket Fortran tilbyr relativt enkel parallellisering av doble og triple løkker. [Uni06]

Metoder

Videre kan man parallellisere på rekursjon og metodekall. Rekursjon er en programmeringsteknikk som gjør det mulig for en metode å kalle seg selv. På den måten kan det forekomme mange kall på den samme metoden, og ved å la instruksjonssekvensene som hører til hvert metodekall kjøre på en egen prosessor kan man parallellisere dette. PRP (Parallelle Rekursive Prosedyrer) er en metode for å automatisere dette.

Prosesser

Inndeling i prosesser er operativsystemets måte å organisere maskinens oppgaver. Operativsystemet har ansvar for å ha kontroll på maskinens alle grunnleggende funksjoner, slik som tastatur og skjerm, og det er vanlig at det opprettes en prosess for hver av disse nødvendige funksjonene. På samme måte skal operativsystemet håndtere alle programmer som kjøres på maskinen, og for hvert program som startes opprettes en egen prosess. Prosessene er uavhengige, og det er operativsystemets oppgave å la de dele på de ressursene maskinen har til rådighet. Enhver prosess rangeres av operativsystemet og tildeles ressurser med jevne mellomrom avhengig av prosessenes prioritet.

Kapittel 4

JavaPRP

Gjennom PRP-prosjektet har det blitt produsert mange masteroppgaver, og som et resultat av dette har det gradvis blitt utarbeidet et system for parallellisering. Siste versjon av dette systemet heter JavaPRP.

JavaPRP er en et system som tar et program som input og forandrer på den opprinnelige koden slik at det kan kjøres i parallell over mange maskiner. Systemet fungerer som en *preprosessor* som modifiserer kildekoden i det opprinnelige programmet. Det genereres i tillegg nye klasser og mekanismer som sørger for parallelliseringen.

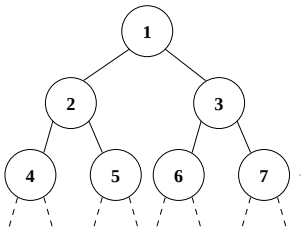
Dette kapittelet forklarer hvordan JavaPRP virker. Senere i denne oppgaven vil det introduseres ett nytt system som er basert på JavaPRP.

4.1 Rekursjon

JavaPRP parallelliserer et program som i utgangspunktet bare kan eksekveres sekvensielt. Det kan ikke være et hvilket som helst program, det må være et program som løser et problem ved hjelp av rekursjon.

En direkte rekursiv metode er en metode som kaller seg selv [Wei99b]. En metode kan også være indirekte rekursiv hvis den kaller en annen metode som igjen kaller den første metoden.

For at en rekursiv metode skal kunne parallelliseres er det viktig at den utfører minst to kall på seg selv. Vi sier at metoden må ha *fanout* minimum lik 2. Dette er for at rekursjonstreet skal vokse i bredden. Et tre med fanout lik 1 vil ikke vokse. Figur 4.1 viser et rekursjonstre med fanout lik 2. Hver node i treet representerer et delproblem. Ideen er at man først bygger opp øverste del av treet til man har mange nok noder i bredden. Deretter kan alle de nederste nodene eksekveres i parallell. Dersom man har et parallelt system med 4 arbeidere ville node 4, 5, 6 og 7 på figur 4.1 kunne eksekveres i parallell ved å bruke denne fremgangsmåten.



Figur 4.1: Rekursjon med fanout lik 2

4.2 Nøkkelord

JavaPRP er ment å være svært enkelt i bruk. En programmerer med dårlig kjennskap til parallellisering skal kunne benytte dette systemet til å parallellisere programmer. Alt brukeren behøver å gjøre er å lage et sekvensielt program og mate dette inn i systemet. JavaPRP vil så generere et parallelt system basert på det sekvensielle programmet.

Før det sekvensielle programmet kan preprosesseres må det imidlertid annoteres med noen faste nøkkelord. Dette er for at systemet skal forstå hvilken del av programmet som skal parallelliseres. Disse nøkkelordene som følger:

- `/* PRP_PROC */`
  Markerer den rekursive metoden. Kommentaren settes inn på linjen rett over metodens header.
- `/* PRP_CALL */`
  Markerer det rekursive kallet. Kommentaren settes inn på linjen over utrykket som inneholder kallet.
- `/* PRP_FF */`
  Brukes dersom det er ønskelig å bruke full fanout. Med full fanout mener vi at alle noder i treet (unntatt rotnoden) er direkte barn av rotnoden.

Listing 4.1: Rekursivt program med nøkkelord

```
1  /* PRP_FF */
   class Eksempel {

     /* PRP_PROC */
5    void rekursivMetode(int param) {

       /* PRP_CALL */
       rekursivMetode(param);
     }
10 }
```

Et program som er klargjort for preprosessering kan se ut som i listing 4.1 på forrige side.

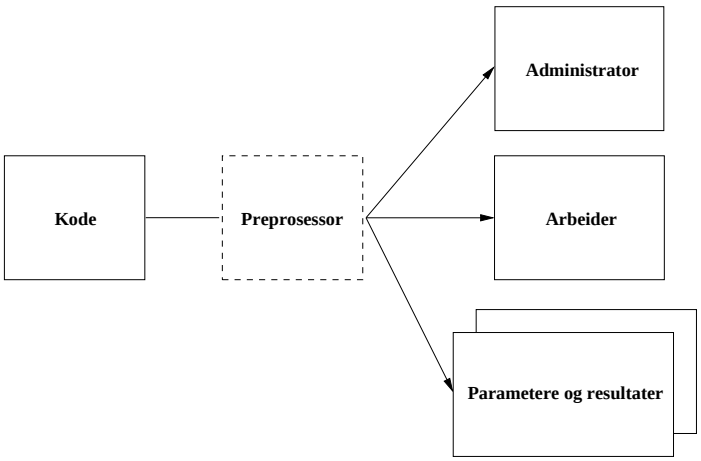
4.3 Administrator/arbeider-modell

PRP-systemet baserer seg på en administrator/arbeider-modell. Dette vil si at en administrator har ansvar for å kalle på de mange arbeiderne. Hver arbeider kjører på en egen maskin i nettverket og løser delproblemer den får tilsendt av administratoren. Følgende klasser benyttes:

- **Arbeideren**
  Arbeiderens oppgave er å utføre selve beregningene der den benytter seg av de parametersett den får tilsendt. Instanser av arbeiderklassen kjører på mange maskiner på nettverket.

- **Administratoren**
  Denne klassen har ansvaret for å håndtere arbeiderne. Den sender parametersett til arbeiderne, og mottar deres svar.

- **Parametersett og returverdier**
  Parametersettene inneholder parametere som arbeiderne kan bruke som input til sine beregninger. Returverdier sendes så tilbake til administratoren.



Figur 4.2: Modell av virkemåten til preprosessoren i JavaPRP

Som vist på figur 4.2 virker JavaPRP ved at en preprosessor leser brukerens kode og produserer nye filer. Filene kompileres og distribueres over nettverket. Alle maskinene som skal benyttes tildeles en arbeider som må

startes. Administratoren kjøres på en av maskinene. Parametersett og returverdier distribueres via en webserver under kjøring.

Kommunikasjonen mellom administratoren og arbeiderne foregår ved hjelp av Java Remote Method Invocation, eller JavaRMI. Dette er et bibliotek i Java. Funksjonaliteten ligner på Remote Procedure Calls (RPC) som ble benyttet i en tidligere versjon av PRP. JavaRMI tillater programmer å utføre kall på metoder i programmer som kjører på andre maskiner.

Etter at alle filer er distribuert og startet opp vil man ved hjelp av administratoren sette i gang selve kjøringen av det genererte systemet. Under kjøringsprosessen går administratoren gjennom tre faser: parametergenerering, kodeeksekvering og svargenerering.

4.4 Parametergenerering

Under parametergenereringen benyttes to datastrukturer: en FIFO-kø (First In First Out) og en kall-stack som begge inneholder parametersett.

- **FIFO-kø**
  Dette skal bli en liste med parametersett som representerer delproblemer som arbeiderne kan løse i parallell.

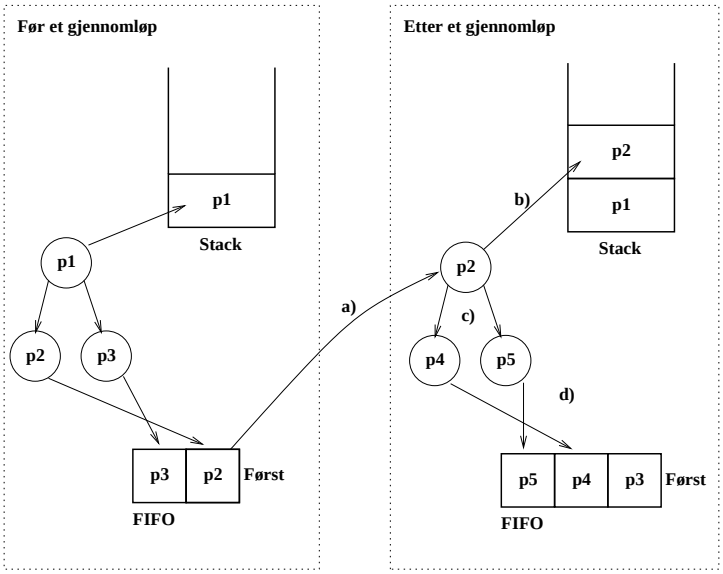
- **Stack**
  Her legges parametersett knyttet til øverste del av rekursjonstreet, altså det som ikke parallelliseres. Disse vil benyttes under oppsamling av resultater senere.

Målet med parametergenereringen er å fylle FIFO-køen med tilstrekkelig mange parametersett. Senere, under eksekvering, vil disse sendes til arbeiderne for å løses.

Ved oppstart legges første parametersett i FIFO-køen. Dette er det opprinnelige parametersettet til den opprinnelige rekursive metoden. Figur 4.3 på neste side viser hvordan parametergenereringen fungerer. Administratoren løper gjennom en løkke som går gjennom fire steg:

1. Popper første parametersett fra køen (skritt a)
2. Pusher dette på stacken (skritt b)
3. Genererer nye parametersett ved å ta utgangspunkt i parametersettet som nettopp er hentet ut (skritt c). Dette gjøres ved å kjøre den opprinnelige rekursive koden frem til kall-stedet der man i stedet for å gjøre kallet oppretter nye parametersett med de aktuelle verdiene.
4. Legger de nye parametersettene bakerst i køen (skritt d)

Ved å repetere disse fire stegene vil FIFO-køen fylles med parametersett. Dette gjøres inntil tilstrekkelig mange parametere er generert.



Figur 4.3: Figuren viser ett gjennomløp av løkken som genererer parametere. p1, p2 osv. representerer parametersett. Løkken utfører fire skritt som legger parametere på stacken og FIFO-køen.

4.4.1 Lastbalansering

I JavaPRP genererer man minst 20 ganger så mange parametere som antall arbeidere. Grunnen til at man genererer så mange parametersett er at man kan risikere at noen delproblemer er større enn andre. Det kan føre til at noen arbeidere får mye mer å gjøre enn andre. Ved å generere ekstra mange og små delproblemer vil man redusere risikoen for at dette vil inntreffe.

4.5 Kodeeksekvering

Under eksekvering vil administratoren sende parametersett til arbeiderne. Arbeiderne vil løse delproblemer basert på parametersettene ved hjelp av rekursjon. Når en arbeider er ferdig med et delproblem vil administratoren motta svar som legges i en array. Arbeideren er nå klar til å begynne på et nytt delproblem.

For å umiddelbart kunne sette i gang med løsing av det neste delproblemet sørges det for at arbeideren allerede har dette liggende klart når den er ferdig med det forrige. Dermed unngår man unødvendig ventetid.

4.6 Svargenerering

Når all eksekvering er ferdig gjenstår oppsamlingen av resultatene. Til dette bruker administratoren en svar-array samt kall-stacken. Administratoren går igjennom en løkke som utfører følgende tre skritt:

1. Administratoren henter ut det øverste elementet på stacken og kjører den rekursive metoden med dette parametersettet frem til kall-stedet.

2. Det er nå ikke nødvendig å gjøre det rekursive kallet da dette delproblemet allerede er løst av arbeiderne. I stedet hentes resultatet ut fra svar-arrayen.

3. Den resterende koden i den rekursive metoden kjøres slik at et nytt svar genereres. Dette legges også i svar-arrayen.

Deretter går administratoren videre med skritt 1 og gjentar prosessen. Den henter på ny ut øverste element på stakken, og det hele gjentar seg inntil stakken er tom.

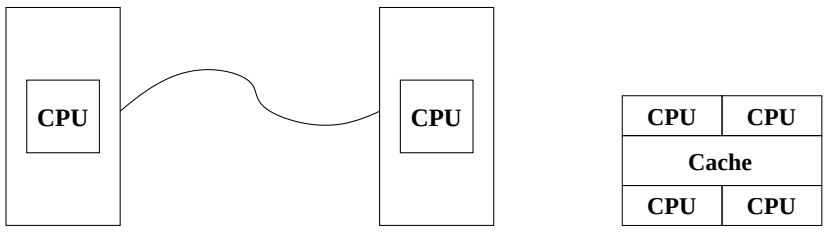
Det siste parametersettet på stacken er argumentet til den opprinnelige rekursive metoden. Dette vil dermed generere det endelige svaret på det opprinnelige problemet.

Kapittel 5

Multiprosessor

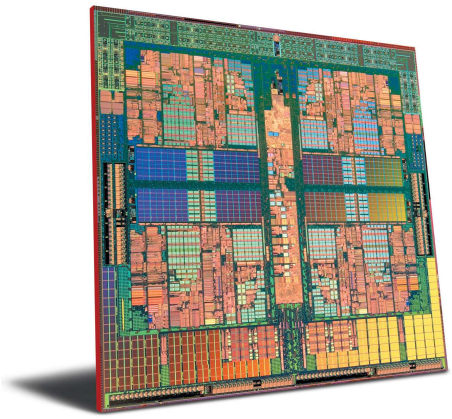
PRP-prosjektet har i alle år kun dreiet seg om parallellisering på maskiner tilknyttet et nettverk. Denne oppgaven vil gå ett skritt videre og fokusere på multiprosessorer. Maskiner av typen multiprosessor bør kanskje håndteres annerledes enn de maskinene PRP så langt har blitt benyttet på. Det vil derfor undersøkes hvordan parallellisering på multiprosessor best mulig bør utføres.

I dette kapittelet blir det gitt en introduksjon til hvordan en multiprosessor er bygd opp og virker. Dette vil gi et utgangspunkt for hvordan programmer for multiprosessor bør konstrueres for at de skal fungere tilfredsstillende.



(a) Nettverk av maskiner (b) Multiprosessor

Figur 5.1: Multiprosessor sammenlignet med nettverk av maskiner. Prosessorene i multiprosessoren kan kommunisere raskt ved hjelp av delt cache/minne, mens maskinene i nettverk må sende data mellom hverandre over store avstander.



Figur 5.2: Bildet viser den nylig lanserte multikjerneprosessoren AMD Phenom X4. Prosessoren har fire kjerner på samme brikke med delt L3-cache. [Dev08]

5.1 Tettere tilknytning

En multiprosessor er en datamaskin med flere prosessorer som er knyttet tett sammen. Dette kan karakteriseres som et fast koblet system. Maskinen kan for eksempel være bygd opp ved at mange prosessorer er montert på samme hovedkort eller i samme serverrack.

En annen variant er en såkalt multikjerneprosessor. Da er mange prosessorkjerner satt sammen på en og samme brikke som vist på figur 5.1(b) på forrige side. En multikjerne-prosessor kan tilsynelatende se ut som en vanlig enkeltkjerneprosessor, men består altså av mange slike enkeltkjerneprosessorer som sitter tett sammen. Figur 5.2 viser den nyeste multikjerneprosessoren fra AMD som er en av mange ulike modeller på markedet. Det finnes multikjerneprosessorer med opp til 11 kjerner, og det er gjort forsøk med opp til 80 kjerner på en brikke [PCW07].

Den største forskjellen mellom parallellisering på multiprosessor og pallellisering over nettverk er nettopp det at i multiprosessoren sitter prosessorene tettere sammen. Det gjør at prosessorene kan kommunisere raskere, noe som ofte er viktig ved parallellisering. Nøyaktig hvor stor avstand det er mellom de ulike prosessorene i et slikt system kan ha stor betydning for ytelsen. En multikjerneprosessor der prosessorene sitter helt inntil hverandre er det klart mest fordelsaktige, fordi kommunikasjonen går raskere jo tette sammen de sitter.

Det finnes flere måter å la prosessorene kommunisere på. En måte er meldingsutveksling hvor prosessorene sender beskjeder til hverandre. Meldingsutveksling kan benyttes på de fleste typer parallelle systemer. På en multiprosessor kan man i tillegg la prosessorene benytte seg av delt hu-

kommelse. Det kan ofte være en stor fordel innenfor parallellisering. Senere i dette kapittelet vil det bli gått nærmere inn på delt hukommelse, men først kommer en oversikt over hvordan hukommelseshierarkiet på en vanlig datamaskin er bygd opp.

5.2 Datamaskinarkitektur

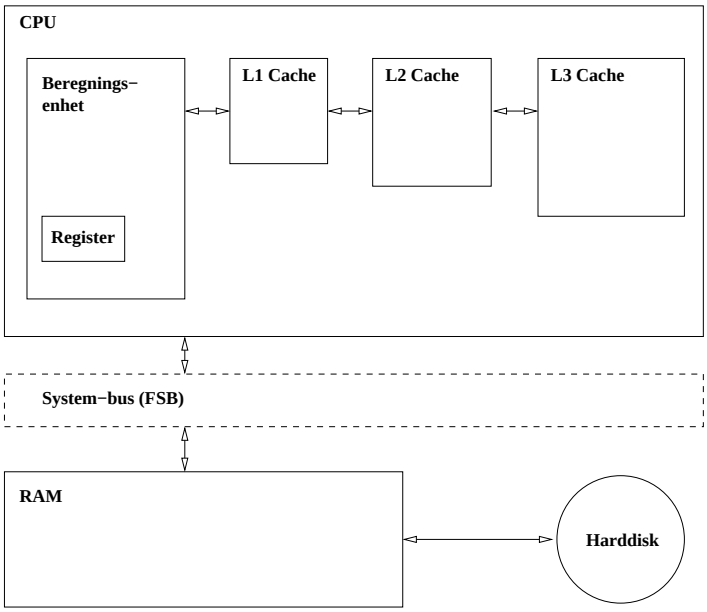
En datamaskin benytter seg av RAM (random access memory) som hovedlager. Programmer benytter dette lageret til å lagre aktuelle data de bruker under sine beregninger. Programmene kan lese fra og skrive til minnet etter behov.

I RAM lagres dataene på minnebrikker, noe som gir relativt kort aksesstid. Til sammenligning tar det omtrent tusen ganger lenger tid å lese data fra en harddisk. Harddisker har imidlertid mye større lagringsplass og brukes derfor vanligvis for lengere tids lagring av permanente data. Hastigheten til harddiskene har dessuten nesten ikke økt. Dagens harddisker har aksesstid på omtrent $5ms$ [Dig08] mens en harddisk i 1973 hadde aksesstid på $25ms$ [IBM08].

Tidligere var RAM raskt nok til å holde følge med CPU, noe som gjorde at CPU ikke behøvde å vente på data som den ønsket å lese fra RAM. Men etter hvert som CPU har blitt raskere har aksesstiden til RAM vist seg å bli en flaskehals [MV99]. Dersom en moderne prosessor utelukkende kunne benytte seg av vanlig RAM ville den ha kastet bort mye tid på å vente hver gang data skulle hentes ut av minnet. Dette problemet har blitt løst ved å innføre flere nivåer av minne i form av cache.

Cache er en type hukommelse som ligger på samme brikke som prosessoren. Dette gjør at kommunikasjonen mellom prosessoren og cache kan gå ekstremt fort. Ulempen er at cache ikke kan ha spesielt stor lagringskapasitet fordi den skal få plass på en så liten brikke, og fordi cache er mer kostbar enn andre typer hukommelse. På dagens prosessorer finner man opp til tre nivåer av cache i tillegg til prosessorens register.

Det høyeste nivået ligger nærmest CPU. Dette nivået har raskest aksesstid og kalles nivå 1 (L1). De påfølgende nivåene ligger lenger ut og har lengre aksesstid, men større plass. (Se figur 5.3) Dette systemet fungerer ved at data som er aktuelle for prosessoren plasseres i et nivå i nærheten av den, mens mindre aktuelle data plasseres lengre ut. Det samme gjelder også for organisering av data i RAM og på harddisken. Når prosessoren skal hente ut data letes det først i L1. Finnes det ikke der sjekkes det på lavere nivåer inntil dataene blir lokalisert.



Figur 5.3: Typisk datamaskinarkitektur. Figuren viser de ulike typer hukommelse man finner i datamaskinen, og deres relasjoner.

5.3 Delt hukommelse

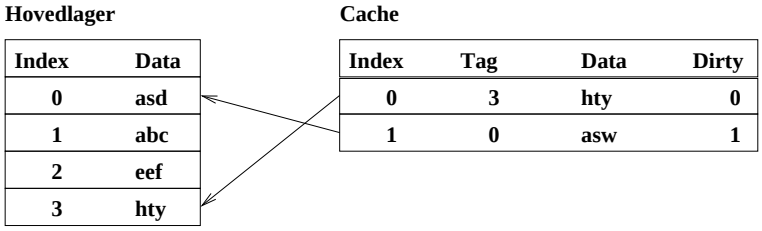
Prosessorene i en multiprosessor har ofte delt minne. Det vil si de er tilknyttet samme hovedlager (RAM) og kan derfor lese og skrive til de samme dataene. Dette er noe man ikke har mulighet til dersom man parallelliserer over nettverk.

5.3.1 Cache

En multikjerneprosessor kan i tillegg ha mulighet til å operere med delt cache. Cache er en form for hurtigminne som gjerne er plassert på samme databrikke som prosessoren. Den korte avstanden mellom prosessoren og cache gjør at lesing og skriving til cache går svært raskt.

Sammenlignet med RAM er det plass til relativt lite data i cache. Derfor sørger et system i prosessoren for at kun de mest brukte dataene ligger der. Systemet kopierer blokker, eller cachelinjer, fra hovedlageret inn i cache ved behov. På figur 5.4 på neste side er to linjer kopiert inn i cache fra hovedlageret.

Når prosessoren skal lese fra minnet letes det først etter de aktuelle dataene i cache. Hvis de ikke ligger der blir en cachelinje med data (typisk 8 til 512 Byte) kopiert fra RAM og lagt i cache. Cachelinjen inneholder det prosessoren leter etter, samt andre data som tilfeldigvis ligger i samme linje.



Figur 5.4: Forholdet mellom cache og hovedlageret. Cache inneholder en kopi av noen utvalgte cachelinjer fra hovedlageret. Tallet i 'Tag'-kolonnen indikerer hvilken blokk i hovedlageret cachelinjen er en kopi av. Linjen med index 1 i cachetabellen er dirty (skitten/modifisert) og innholder andre data enn den tilsvarende linjen hovedlageret. Linjen må derfor før eller senere kopieres tilbake til hovedlageret.

Hvis cache er full vil de cachelinjene som er minst brukt bli overskrevet.

Når prosessoren har skrevet noe til cache må dataene før eller senere også kopieres videre til hovedlageret. På figur 5.4 har prosessoren skrevet i cachelinjen med index 1 i cache. Denne vil bli kopiert videre til hovedlageret, men på hvilket tidspunkt dette blir gjort kan styres på forskjellige måter.

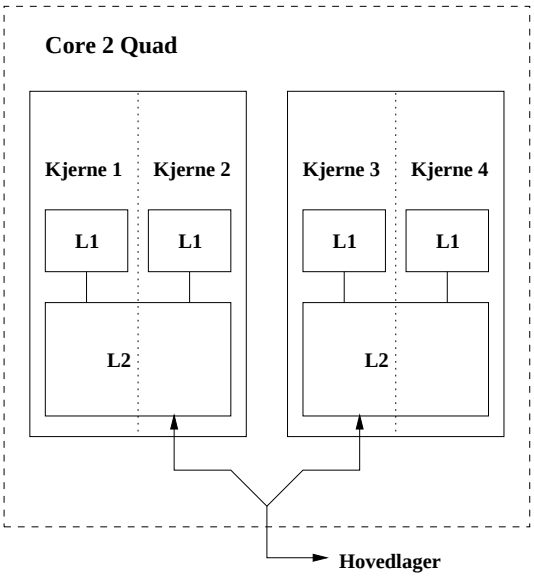
Dersom man opererer med **write-through** blir alt som skrives til cache umiddelbart også skrevet til hovedlageret.

Ved bruk av **write-back** er ikke dette tilfellet. I stedet blir cachelinjer som det skrives til markert som **dirty** (skitten/modifisert). Disse behøver ikke å bli kopiert tilbake til hovedlageret før det er nødvendig. Det kan være hvis det skal hentes inn en cachelinje fra hovedlageret når cache er full. Da må eventuelt en cachelinje som er dirty først kopieres til hovedlageret før den overskrives.

5.3.2 Delt cache

Delt cache gir prosessorene enda mer effektiv bruk av delt hukommelse. For å kunne ha delt cache må de enkelte prosessorene i multikjerneprosessoren være del av en og samme databrikke. Hvis man da også har cache på denne brikken kan de ulike prosessorene være tilknyttet samme cache.

Figur 5.5 på neste side viser cachearkitekturen til multikjerneprosessoren som benyttes i forsøkene i dette prosjektet. Denne består egentlig av



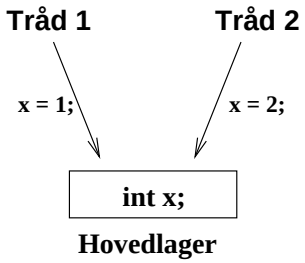
Figur 5.5: Cachearkitektur på Intel Core 2 Quad (Xeon E5320)

2 multikjerneprosessorer montert på samme sokkel. Hver enkelt av disse to multikjerneprosessorene består av en databrikke som inneholder 2 prosessorkjerner. Til sammen har man da 4 prosessorkjerner. Her har alle prosessorene sin egen L1-cache, mens de to som ligger på samme brikke deler L2-cache. Man har altså et system med delvis felles cache, der hvert prosessorpar kan dele data svært effektivt. Hvis alle fire prosessorene skal dele data mellom hverandre må man ut til hovedlageret (RAM).

5.4 Bruk av felles variabler

Et program som benytter parallelle ressurser gjør dette gjerne ved å opprette et antall tråder som kan kjøre i parallell. Trådene kan ha tilgang til felles hukommelse gjennom å ha tilgang til felles metoder eller felles variabler. Ved bruk av delte data er det imidlertid en del ting man må være oppmerksom på for at programmet skal fungere som det skal.

En tråd kan bruke delte data på to måter. Den kan lese eller den kan skrive. Begge situasjonene kan føre til problemer av ulik art. Reglene for hvordan tråder kan benytte felles hukommelse er definert i Java Memory Model (JMM) [GJSB00]. Dersom flere tråder forsøker å skrive til samme data samtidig kan det oppstå såkalte **race conditions**. Ved lesing fra delte data kan man på grunn av **asynkron cache** risikere å ikke motta siste oppdaterte verdi.



Figur 5.6: Race conditions. Tråd 1 og tråd 2 forsøker i parallell å skrive til variabel x. For å kjenne resultatet må man kontrollere rekkefølgen trådene får skrivetilgang til variabelen.

5.4.1 Race conditions

Dersom en tråd skriver til felles data er det viktig å passe på at ingen andre tråder forsøker å benytte seg av dataene mens skrivingen foregår. Om flere tråder i parallell forsøker å skrive til samme minnelokasjon kan man ikke vite hvilken verdi som til slutt ender opp i minnelokasjonen.

I slike situasjoner vil mekanismer i datamaskinen og operativsystemet styre hvordan ressurser deles mellom parallelle tråder. I virkeligheten er det ikke mulig at flere tråder kan aksessere felles data nøyaktig samtidig. Det som skjer er at noen tråder må vente på tur til andre er ferdige. Problemet er at man ikke vet hvilken tråd som får slippe til først. Man vet derfor ikke hvilke data som til slutt blir liggende i den felles hukommelsen, og man vet ikke hvilke data som eventuelt senere leses av de ulike trådene.

Slike problemer med bruk av felles variabler kalles race conditions. Det er viktig å unngå slike race conditions, ettersom det kan gi uforutsette og feilaktige resultater.

5.4.2 Asynkron cache

Når et program benytter seg av mange tråder deler disse hovedlager og adresseområde. Men som vi har sett tidligere er det ikke nødvendigvis slik at de deler cache. Det kan i grunn antas at alle trådene kjører på en separat prosessor, der alle prosessorer har tilgang til samme hovedlager, men hvor alle har sin egen cache. Cache er ikke nødvendigvis alltid synkronisert med hovedlageret. Figur 5.4 på side 25 viser et eksempel på dette.

Dersom to tråder har en referanse til en variabel i hovedlageret er det i følge Java Memory Model mulig at de begge har en lokal variant av variabelen i sin egen cache som ikke nødvendigvis er lik den i hovedlageret. [Goe01]

5.4.3 Synkronisering

En strategi for å unngå både race conditions og asynkron cache kan være synkronisering.

I Java har man to synkroniseringsmekanismer: (1) Synkroniserte metoder (og blokker) og (2) volatile variabler.

Synkroniserte metoder

Synkroniserte metoder er metoder som kun én tråd kan kjøre om gangen. Dersom to tråder kaller metoden samtidig må den ene vente på tur til den andre er ferdig med å eksekvere metoden.

Dette gjøres ved hjelp av såkalte låser. Når en tråd går inn i metoden låser den en lås knyttet til metoden. Når den forlater metoden låses den opp. Så lenge låsen er låst kan ingen andre tråder gå inn i metoden.

Inne i den synkroniserte metoden kan tråden utføre oppgaver som den må være alene om å utføre. Et eksempel på en slik oppgave er skriving til felles variabler. Hvis all skriving skjer inne i en synkronisert metode kan man garantere at kun én tråd skriver om gangen.

Synkroniserte metoder løser også problemet med asynkron cache. Når en tråd låser en lås krever JMM at trådens cache må synkroniseres med hovedlageret. Dermed vet man at de nyeste dataene brukes. Dessuten kreves det også at alt endret innhold i cache blir flushet (kopiert tilbake tilhovedlageret) umiddelbart etter at låsen frigjøres. [Goe01]

Dette er også en demonstrasjon av hvorfor synkronisering kan påvirke programmers ytelse og en god grunn til av dette bør brukes så sjelden som mulig. Det å flushe cache kan ta mye tid.

Listing 5.1 viser eksempel på en synkronisert metode for modifisering av en delt variabel. Mange tråder kan kalle metoden, men kun én får slippe til om gangen og faktisk eksekvere kodelinjene.

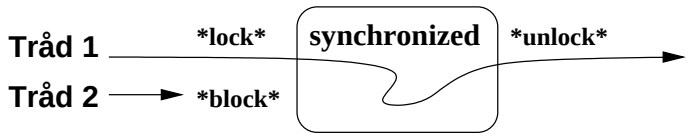
Listing 5.1: Synkronisert metode

```
1  synchronized void setValue(int newValue) {
     this.value = newValue;
   }
```

Synkroniserte blokker

Synkroniserte blokker er omtrent det samme som synkroniserte metoder, bortsett fra at det her er innholdet i en blokk som markeres som synkronisert.

Til forskjell fra synkroniserte metoder kan man bestemme hvilken lås som skal benyttes ved synkroniseringen. Ved bruk av synkroniserte metoder brukes alltid en lås som er knyttet til klassen som metoden befinner seg i. Alle synkroniserte metoder i klassen bruker denne ene låsen.



Figur 5.7: Synkronisering. Tråd 1 og tråd 2 forsøker å gå inn i den synkroniserte metoden / blokka. Tråd 1 slipper til først. Ved inngang låses en lås. Det gjør at ingen andre slipper inn. Ved utgang låses låsen opp igjen. Tråd 2 måtte blokkere da låsen var låst. Etter at låsen låses opp kan den slippe inn i metoden.

Ved deklarasjon av en synkronisert blokk sender man med den låsen man vil skal benyttes. Låsen kan være initialisert tidligere i koden. Ved inngang til blokka i listing 5.2 låses en slik egendefinert lås.

Listing 5.2: Synkronisert blokk

```
1  myValue = x;
   synchronized (valueLock) {
     publicValue = myValue;
   }
```

Volatile variabler

Nøkkelordet `volatile` kan benyttes i Java til å markere variabler som skal brukes av flere tråder samtidig. Listing 5.3 viser hvordan en variabel markeres ved deklarasjonen.

Dette nøkkelordet sikrer at variabelens verdi alltid hentes fra hovedlageret ved lesing, og alltid skrives til hovedlageret ved skriving. Dette gjør at man unngår problemet med asynkron cache som fører til feilaktige kopier lokalt i cache eller registeret.

Lese- og skriveoperasjoner på volatile variabler kan anses for å være atomiske operasjoner.

Listing 5.3: Bruk av volatile variabler

```
1  volatile int value;
```

5.4.4 Atomiske operasjoner

En annen måte for å håndtere felles hukommelse er å bruke atomiske operasjoner [Mic08]. En atomisk operasjon er noe en tråd kan gjøre uten at den kan bli avbrutt av andre tråder.

Eksempel på atomiske operasjoner er lese- og skriveoperasjoner på volatile variabler. Linje 4 i listing 5.4 på neste side viser eksempel på en slik operasjon.

På linje 7 vises en operasjon som ikke er atomisk. Det er fordi denne operasjonen består av flere instruksjoner. Først leses verdien, deretter økes den med en, og så skrives resultatet tilbake til minnet. Under denne operasjonen kan tråden bli avbrutt av en annen tråd som også skriver til variabelen. Dermed kan man ikke garantere hva utfallet blir.

Listing 5.4: Atomiske og ikke-atomiske operasjoner

```
1  volatile int value;

   //atomisk operasjon
   value = 3;
5
   //ikke-atomisk operasjon
   value++;
```

Java 5 tilbyr et eget klassebibliotek for atomiske operasjoner [Mic04]. I pakken finner man en rekke klasser som kan brukes som erstatning for datatyper i Java. For eksempel har man `AtomicInteger` som inneholder lagring av et heltall og metoder for modifisering av dette. Innholdet i disse metodene er atomiske operasjoner, noe som for eksempel sikrer at flere tråder kan lese fra samme data i parallell.

I tillegg til å hindre race conditions sørger de atomiske datatypene for å unngå asynkron cache. Dette gjøres ved at deres innhold alltid lagres i hovedlageret, på samme måte som volatile variabler.

Listing 5.5 viser eksempler på noen av disse datatypene.

Som vist inneholder pakken også klasser for atomiske arrayer. (feks. `AtomicIntegerArray`). Denne gir en unik mulighet til å la flere tråder operere på ulike deler av samme array samtidig.

Alternativet er å benytte en vanlig array. Men innholdet i vanlige arrayer kan ikke markeres som volatile, så løsningen må da være å benytte synkronisering, altså å låse hele arrayen ved bruk. Men da kan man ikke la flere tråder bruke deler av arrayen samtidig. Det går med atomisk array.

Listing 5.5: Bruk av pakken atomic i Java

```
1  import java.util.concurrent.atomic.*;
   ...

   AtomicInteger value;
5  AtomicDouble value2;
   AtomicIntegerArray values;
   ...

   value.set(3);
10 values.set(1, 5);//setter element 1 lik 5
```

5.5 Parallellisering på multiprosessor

En multiprosessor har visse egenskaper som påvirker hvordan man kan utføre parallellisering på denne.

For det første har man tilgang til alle de parallelle ressursene på et sted, i en maskin. Det gjør at multiprosessoren trolig er enklere å håndtere under både bruk og utvikling av det parallelle systemet. På en multiprosessor kan man kjøre parallelle systemer som kun består av et enkelt program som selv tar seg av parallelliseringen ved hjelp av tråder. Dette er enklere enn for eksempel de parallelle systemene som genereres av JavaPRP, der mange programmer må kjøres samtidig på mange maskiner.

En annen fordel med multiprosessor er delt hukommelse. Som beskrevet tidligere byr imidlertid bruk av dette på en del utfordringer.

5.5.1 Eksempelprogrammer

Fordi parallellisering på multiprosessor skiller seg fra hvordan man parallelliserer på maskiner tilknyttet nettverk vil jeg ta for meg noen eksempler på hvordan dette kan gjøres. Det vil bli beskrevet tre ulike parallelle programmer. De tre programmene skiller seg fra hverandre ved at den parallelle delen er konstruert på tre forskjellige måter med opphav i tre ulike typer sekvensielle algoritmer.

Målet med dette er å finne ut hvordan parallellisering på multiprosessor kan utføres på best mulig måte. Deretter vil det sees nærmere på automatisk parallellisering, og da vil målet være å oppnå programmer som fungerer like godt som programmer som er parallellisert manuelt.

- **Kapittel 6: Rekursjon**
  Kapittelet vil presentere en rekursiv algoritme for løsing av Traveling Salesperson. Det vil så bli presentert en parallell variant av den samme algoritmen, som er inspirert av mekanismer benyttet i JavaPRP. De parallelle trådene i programmet benytter seg av delte data ved skriving til og lesing fra en felles variabel.

- **Kapittel 7: Løkke**
  Kapittelet presenterer en algoritme for løsing av Goldbachs hypotese. Algoritmen benytter seg av en løkke som vil parallelliseres i en parallell variant av programmet. De parallelle trådene leser og skriver til en felles array, men til helt ulike elementer.

- **Kapittel 8: Gjentatt løkke / pumpe**
  Til slutt presenteres et program for generering av optimalt søketre. Algoritmen har en dobbel løkke. Det vil så presenteres flere ulike programmer som parallelliserer dette. Algoritmen er spesiell fordi det kun er den indre løkken som kan parallelliseres. Den ytre løkken må

være sekvensiell også i den parallelle utgaven. For å løse dette kan synkronisering benyttes, og de ulike programmene som foreslås benytter ulike teknikker for dette.

5.6 Testmaskin

Forsøkene i denne oppgaven er gjort på en maskin med prosessor av typen Intel Xeon E5320. Det er en 64bits-prosessor med klokkefrekvens på 1.86GHz. Prosessoren har fire kjerner som vist på figur 5.5 på side 26. Den er utstyrt med 4MB L2-cache for hvert kjernepar, slik at total mengde L2-cache er 8MB.

Prosessoren har også støtte for såkalt Hyper-Threading [Wik08a]. Det er en teknologi som gjør at hver prosessor under visse omstendigheter er i stand til å eksekvere to tråder i parallell like effektivt som ved bruk av to separate prosessorer. Hyper-Threading virker ved at visse kretser på prosessoren er duplisert. For operativsystemet kan prosessoren utgi seg for å være to logiske prosessorer. I dette tilfellet gir det altså operativsystemet mulighet til å eksekvere 8 tråder i parallell effektivt. De parallelle programmene i dette prosjektet bør derfor kunne benytte seg av minimum 8 tråder.

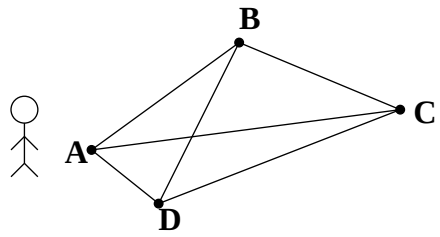
Operativsystemet på maskinen er x86_64 GNU/Linux 5.2.1.

Kapittel 6

Problem 1: Travelling Salesperson - et rekursivt problem

The Travelling Salesperson (Den reisende selger) [Wei99d] er et kjent problem innenfor kompleksitetsteori. Problemet tilhører klassen NP-hard, og det innebærer at dette er et tidkrevende problem å løse.

Problemet lyder som følger: Gitt et antall byer og kostnadene for å reise mellom alle par av byer, hva er den rundturen med minst kostnad som besøker alle byer én gang og returnerer til start? Figur 6.1 illustrerer problemet.



Figur 6.1: Travelling Sales Person: Dersom en person starter ved punkt A og avstanden mellom alle par av punkter er kjent, hva er den korteste ruten personen må reise for å besøke alle punktene og returnere til A?

6.1 Sekvensiell algoritme

Det er utviklet mange effektive algoritmer for dette problemet. For ikke å gjøre ting for komplisert under parallellisering sees det her kun på en forholdsvis enkel algoritme. Algoritmen benytter seg av rekursjon. Senere blir det presentert en parallell variant av denne algoritmen.

En intuitiv måte å løse problemet på er å prøve alle mulige ruter gjennom grafen. Den korteste ruten er svaret på problemet. Algoritmen som blir brukt her søker seg gjennom grafen rekursivt og prøver på den måten alle ruter.

Byer representeres som noder, og nettverket av byer som en graf. En tom array opprettes for å merke noder som besøkte. Programmets rekursive metode kalles med grafens startnode samt besøkt-arrayen som argument. Listing 6.1 viser metoden `tspsolver()` som har nodens `id`, en boolsk besøkt-array og hittil avlagt rute som parametere. Metoden markerer i arrayen at noden er besøkt. Derfra går den alle mulige veier ved å kalle seg selv for alle naboene til noden. En kopi av besøkt-arrayen sendes med slik at det huskes hvilke noder som tidligere er besøkt. Alle nabonodene søker så videre ved å kalle seg selv for alle sine nabonoder som ikke allerede er besøkt. Når alle noder er besøkt har man funnet en mulig rute. Rutens lengde noteres, og dersom den er den korteste ruten man hittil har funnet lagres den.

Listing 6.1: Rekursiv metode for løsing av Travelling Salesperson

```
1  public int tspsolver(int node, boolean[] visited, int path) {
     //hvis søket sålangt er lenger enn cutoff
     if (path >= cutoff) { return -1; }

5    //marker noden
     visited[node] = true;
     int shortestRes = MAX_VALUE;
     //for alle ubesøkte naboer
     for (int i = 0; i < problem.length; i++) {
10     if (!visited[i]) {
         //rekursivt kall
         int tempres = tspsolver(i, visited.clone(), path
         + dist(node, i)) + 1;
         if (tempres < shortestRes && tempres != -1) {
15         shortestRes = tempres;
         }
       }
     }
     //hvis besøkt alle noder
20   else if (foundCompletePath()) {
       updateBestResult(); //cutoff = bestResult
       return problem[node][0];
     } else { return shortestRes; }
   }
```

For å gjøre algoritmen noe mer effektiv benyttes det en cutoff-verdi lik lengden på den korteste ruten som hittil er blitt funnet. I programmet i listing 6.1 på forrige side representeres denne som variabelen `cutoff`. Den brukes til å avskjære søk som viser seg å måtte bli lenger enn det hittil korteste resultatet. Dersom man under et søk finner ut at ruten så langt er lenger enn `cutoff` avsluttes det aktuelle søket.

6.2 Parallell algoritme

Et parallell variant av dette programmet er et program som utfører flere søk i parallell. Programmet må altså opprette mange tråder som i parallell søker gjennom grafen i forskjellige retninger.

Programmet gjør dette ved å benytte seg av arbeidere. Arbeiderne vil kunne løse oppgaver, eller delproblemer, som er generert på forhånd.

Programmet går gjennom tre faser. Først må problemet deles opp i delproblemer. Dette blir under kalt for arbeidergenerering. Så vil alle arbeiderne eksekvere sine delproblemer. Til slutt foretas innsamling av resultatene. Det hele kan minne om hvordan JavaPRP foretar parallellisering.

6.2.1 Arbeidergenerering

Delproblemene vil her bestå av søk gjennom grafen i ulike retninger. For å generere slike delproblemer vil programmet ta utgangspunkt i startnoden og bygge et tre via nodens naboer. Dette er det samme som å bygge opp øverste del av rekursjonstreet til den sekvensielle algoritmen. Treet representerer mulige veier man kan følge gjennom grafen. Programmet vil la dette treet vokse inntil antall bladnoder er tilstrekkelig stort.

For hver bladnode kan man fortsette et søk gjennom grafen og finne en løsning på problemet. Bladnodene representerer derfor delproblemer, og det vil nå opprettes en arbeider for hvert av disse. Arbeideren skal fortsette søket gjennom grafen fra sin egen utgangsposisjon. Det genereres delproblemer inntil man har fem ganger så mange delproblemer som antall prosessorer på maskinen. Det kan være fornuftig å lage flere arbeidere enn antall prosessorer, da noen oppgaver tar kortere tid å løse enn andre. På den måten kan for eksempel én prosessor benyttes til å løse mange små oppgaver mens en annen holder på med én stor oppgave.

6.2.2 Eksekvering

For å utføre beregningene i parallell opprettes et antall tråder t tilsvarende antall prosessorer på maskinen. Pakken `java.util.concurrent` benyttes til dette. Vi har nå t tråder, og $t * 5$ oppgaver som skal løses. En klasse i pakken `concurrent` benyttes til å eksekvere de mange arbeiderne i parallell

ved hjelp av trådene på en slik måte at t arbeidere til en hver tid eksekveres mens andre venter på tur.

Arbeiderne fortsetter et søk gjennom grafen fra sin utgangsposisjon. Dette søket kan gjøres omtrent på samme måte som søket i den sekvensielle algoritmen. Forskjellen er at vi nå har mange arbeidere som gjør dette i parallell.

Alle arbeiderne vil til slutt komme frem til et svar som er den beste veien de hadde mulighet til å finne fra sin utgangsposisjon. Den arbeideren som har det beste svaret har funnet løsningen på hovedproblemet.

6.2.3 Cutoff

I den sekvensielle algoritmen ble det brukt en cutoff-verdi til å avbryte søk som måtte bli lenger enn den hittil beste løsningen. En slik cutoff-verdi er det også ønskelig å benytte i den parallelle utgaven.

Det finnes imidlertid flere muligheter for hvordan denne kan implementeres. Vi kan la hver arbeider ha sin egen lokale cutoff som den alene skal forholde seg til, eller vi kan la alle arbeiderne benytte seg av en felles struktur.

Det sistnevnte er det mest ideelle. Det innebærer imidlertid at trådene må dele data og da må man håndtere problemene knyttet til det ved hjelp av for eksempel synkronisering. Kanskje vil denne synkroniseringen påvirke ytelsen så mye at lokal cutoff-verdi er mer hensiktsmessig.

For å finne ut dette lager jeg tre varianter av det parallelle programmet: (1) Et som ikke benytter cutoff-verdier, (2) et med global cutoff-verdi og (3) et der hver arbeider har egen lokal cutoff-verdi.

6.2.4 Synkronisering

For varianten med global cutoff velger jeg å la cutoff-verdien representeres ved hjelp av en volatile variabel slik at lesing fra denne kan gjøres trygt.

Ved skriving velger jeg dessuten å benytte synkronisering. I den synkroniserte blokka vil en metode oppdatere variabelen hvis den har funnet en ny løsning som er bedre. Listing 6.2 viser hvordan denne blokka ser ut.

Listing 6.2: Synkronisering ved endring av cutoff

```
1  synchronized (cutoffLock) {
     if(thisResult < cutoff) { cutoff = thisResult; }
   }
```

6.3 Resultater

I tillegg til de tre parallelle programmene har jeg testet to sekvensielle programmer, et med og et uten cutoff. Alle programmene er kjørt på prosjek-

tets testmaskin med 4 kjerner (5.6 på side 32).

6.3.1 Uten cutoff

La oss først sammenligne resultatene fra programmene som ikke benytter seg av cutoff-verdi. Vi har to programmer, et sekvensielt og et parallellt. Det at det ikke benyttes cutoff gjør at arbeiderne i det parallelle programmet aldri skriver til en felles hukommelse.

Tabell 6.1 viser resultatene etter tre testkjøringer av hvert program. Det er benyttet problemer av størrelse 10, 11 og 12.

Kolonne 4 i tabellen viser forholdet mellom programmenes tidsforbruk. Man kan se at det parallelle programmet yter betydelig bedre. Det er 3.4 ganger raskere på de største problemene.

Program	Problemstørr.	Tidsforbruk (ms)	Forb.faktor
Sekvensiell	10	467	
Parallell	10	227	2.1
Sekvensiell	11	3833	
Parallell	11	1161	3.3
Sekvensiell	12	39400	
Parallell	12	11694	3.4

Tabell 6.1: TSP uten cutoff. Sammenligning av sekvensiell og parallell algoritme for Traveling Salesperson uten bruk av cutoff-verdi

6.3.2 Med cutoff

Så går vi over til programmene som benytter seg av cutoff-verdi. I tillegg til det sekvensielle er det testet på to parallelle programmer. Det ene benytter en global cutoff-variabel, mens i det andre har hver arbeider sin egen lokale cutoff-variabel.

Resultatene vises i tabell 6.2 på neste side. Som forventet viste dette programmet seg å være mye raskere enn programmet uten cutoff. Jeg har derfor økt størrelsen på problemene til 16, 17 og 18.

Kolonne 4 i tabellen viser forbedringen i tidsforbruk. Programmet med lokal cutoff viser seg å bruke omtrent dobbelt så lang tid som det sekvensielle. Programmet med global cutoff går imidlertid mellom to og tre ganger så raskt.

Det viser seg altså at bruk av felles hukommelse er helt nødvendig for at parallellisering skal lønne seg i dette tilfellet.

Program	Problemstørr.	Tidsforbruk (ms)	Forb.faktor
Sekvensiell	15	4383	
Parallell, lokal cutoff	15	7134	0.6
Parallell, global cutoff	15	1537	2.9
Sekvensiell	16	17564	
Parallell, lokal cutoff	16	30285	0.6
Parallell, global cutoff	16	5769	3.0
Sekvensiell	17	32256	
Parallell, lokal cutoff	17	120671	0.3
Parallell, global cutoff	17	16802	1.9
Sekvensiell	18	142074	
Parallell, lokal cutoff	18	261442	0.5
Parallell, global cutoff	18	58203	2.4

Tabell 6.2: TSP med cutoff. Sammenligning av sekvensiell og parallell algoritme for Traveling Salesperson med bruk av global eller lokal cutoff-verdi

6.4 Oppsummering - rekursjon

Jeg har nå vist hvordan en rekursiv algoritme kan utføres i parallell. Traveling Salesperson er bare et mange slike problemer som kan parallelliseres på samme måte. Til felles har de at de bruker en rekursiv metode som vist i listing 6.4.

Listing 6.3: Rekursjon

```
1  int solveProblem( ... ) {
     //gjøre beregninger rekursivt
     solveProblem( ... );

5    return svar;
   }
```

Disse problemene kan løses ved å generere en mengde arbeidere som utfører sin egen del av rekursjonstreet. Man må altså først bygge opp øverste del av treet og opprette en arbeider for hver bladnode. Dette minner om teknikken som benyttes i JavaPRP.

Kapittel 7

Problem 2: Goldbach - et iterativt problem med løkke

Goldbachs hypotese [Mat08] er et av de eldste uløste problemer innenfor matematikken. Dette kapittelet tar for seg et program som har som formål å motbevise Goldbachs hypotese i parallell på en multiprosessor.

7.1 Goldbachs hypotese

Hypotesen ble første gang beskrevet i 1742 av Christian Goldbach. Den lyder som følger:

Ethvert partall større enn 2 kan skrives som summen av to primtall.

Å uttrykke et partall som summen av to primtall kalles Goldbachpartisjonen av partallet. Noen eksempler:

$$4 = 2 + 2$$
$$6 = 3 + 3$$
$$8 = 3 + 5$$
$$10 = 3 + 7 = 5 + 5$$

Dette problemet kan løses på to måter. (1) Man kan bevise matematisk at hypotesen er sann, eller (2) man kan motbevise den ved å finne et moteksempel. Ingen har ennå klart å løse dette på den ene eller andre måten. Programmet som presenteres her kan benyttes som et forsøk på det siste. Fremgangsmåten er å forsøke å finne et partall som ikke kan skrives som summen av to primtall.

7.2 Sekvensiell algoritme

Programmet har som oppgave å beregne antall Goldbachpartisjoner for alle partall fra 0 opp til en gitt grense n. Finner programmet et partall uten en goldbachpartisjon er problemet løst og programmet avsluttes.

Eksekveringen er som følger:

- Først beregnes og lagres alle primtall mellom 0 og n.
- Deretter tar man for seg ett og ett partall (*even*).
- For hver av disse går man igjennom alle primtall (*prime*) mindre enn *even*/2.
- Dersom $even - prime$ er et primtall har man funnet en goldbachpartisjon.

Min sekvensielle implementasjon av algoritmen går gjennom alle partall i stigende rekkefølge fra 0 til n, og regner ut antall Goldbachpartisjoner for hvert av dem. Dette gjøres ved hjelp av en løkke.

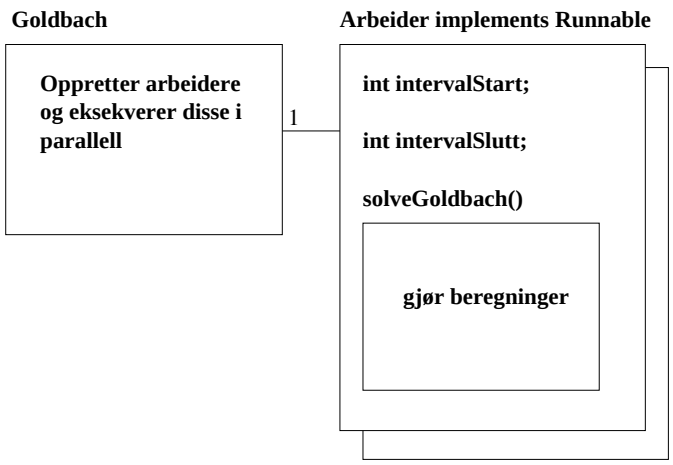
7.3 Parallell algoritme

Utregningen for hvert av partallene kan sees på som et enkeltstående problem. Fordi alle disse problemene er uavhengige er denne algoritmen relativt enkel å parallellisere. Det kan gjøres ved å fordele de mange delproblemene på de ulike prosessorene man har tilgang til i maskinen.

Dette parallelle programmet deler området av partall det skal beregnes for inn i mindre deler. Det lages en oppgave for hver av disse delproblemene. De mange oppgavene representeres som instanser av en egnet klasse som kallers `Arbeider`. Figur 7.1 på neste side viser klassen `Goldbach` sammen med klassen `Arbeider`. `Arbeider` bør ha variabler knyttet til hvilket delproblem som skal løses, samt en metode for løsing av delproblemet. Denne metoden beregner antall goldbachpartisjoner for det intervallet av partall som gjelder for den aktuelle oppgaven. Hver arbeider implementerer interfacet `Runnable` slik at de kan tilordnes en egen tråd og eksekveres i parallell på samme måte som ved eksekveringen i TSP (seksjon 6.2.2 på side 35).

7.3.1 Datastruktur

Programmet lagrer resultatene etter alle beregningene i en array. Dette er en array som skal inneholde antall primtallsummer for hvert partall. Disse partallene, og elementene i arrayen, er uavhengige delproblemer som trådene løser. Trådene skal ha felles tilgang til denne arrayen.



Figur 7.1: Parallell Goldbach

Når man skal la mange tråder benytte seg av en felles datastruktur på denne måten er det viktig at man tar hensyn til problemene knyttet til dette. Man må ta hensyn til race conditions og asynkron cache som er nevnt tidligere.

Fordi elementene i arrayen representerer delproblemer som er uavhengige vil det ikke kunne oppstå race conditions i dette tilfellet. Ett element i arrayen representerer svaret på et delproblem som kun én tråd får i oppgave å løse. Hvert element i arrayen blir derfor bare skrevet til av én tråd.

Når trådene skriver til arrayen kan det hende at de kun skriver lokalt i sin egen cache, og altså ikke skriver til hovedlageret. Som nevnt tidligere kalles dette fenomenet asynkron cache og kan løses ved at cachen flushes ut til hovedlageret. Dette er imidlertid ikke noe problem i dette tilfelle. Så lenge trådene arbeider på problemet utføres det kun skriving til arrayen. Det foregår ingen lesing før trådene er ferdige og avsluttet. I det trådene avsluttes blir imidlertid deres cache synkronisert med hovedlageret, så da vil det bli trygt å lese fra arrayen.

I dette tilfellet er det med andre ord trygt å benytte en vanlig array som felles datastruktur.

7.4 Resultater

Programmet er testet på prosjektets testmaskin (Seksjon 5.6 på side 32). Det har blitt gjort tester der 1, 2, 4 eller 8 tråder benyttes, og med øvre grense fra 10000 til 3000000.

Tabell 7.1 på neste side viser resultatet etter beregningen med øvre grense 3000000. Ved bruk av 8 tråder er det parallelle programmet nesten 7 gan-

Program	Tidsforbruk (s)	Forb.faktor
Sekvensiell	506	
Parallell, 1 tråd	507	1.0
Parallell, 2 tråder	270	1.9
Parallell, 4 tråder	139	3.6
Parallell, 8 tråder	73	6.9

Tabell 7.1: Tidsforbruk for beregning av Goldbachsummer for tall mellom 0 og 3000000. Tabellen viser tidsforbruket til det parallelle programmet ved bruk av ulike antall tråder. Forbedringsfaktoren indikerer forholdet mellom tidsforbruket til det parallelle programmet og det sekvensielle programmet.

ger raskere enn det sekvensielle. Programmet er altså i stand til å utnytte de parallelle ressursene svært godt. Det at programmet har en forbedringsfaktor på 7 ved bruk av en prosessor med 4 kjerner viser at programmet drar fordel av prosessorens funksjonalitet for multithreading.

Man kan også se at det parallelle programmet er omtrent like raskt som det sekvensielle ved bruk av kun 1 tråd. Det tyder på at kostnadene knyttet til selve parallelliseringen er små.

Figur 7.2 på neste side viser forholdet mellom det parallelle programmets kjøretider og det sekvensielle programmets kjøretid for bruk av ulike antall tråder. Grafen viser at forbedringsfaktoren øker jo større problem man skal løse. Med andre ord får man mer nytte av parallelliseringen jo tyngre problem som løses. Ved bruk av 8 tråder får man maksimal forbedringsfaktor (omtrent 7) på problemer med størrelse fra omtrent 1000000.

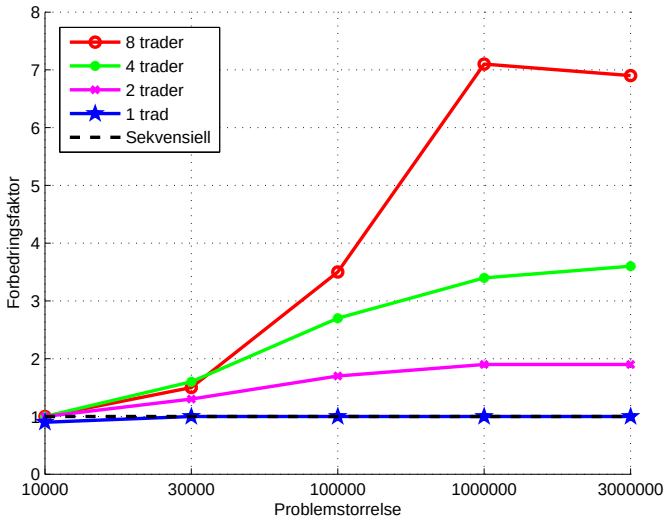
7.5 Oppsummering - løkke

Det har her blitt vist hvordan en algoritme som benytter seg av en løkke kan utføres i parallell. Goldbach er et eksempel på mange slike problemer som kan parallelliseres på samme måte. Til felles har de at problemet består av en lang rekke delproblemer som løses sekvensielt ved hjelp av en løkke som vist i listing 7.5.

Listing 7.1: Løkke

```
1  for(int i=0; i<10000; i++) {
     //Løs dette delproblemet
     solveProblem(i);
   }
```

Disse problemene kan løses ved å generere en mengde arbeidere som hver utfører et utvalg av disse delproblemene. Man kan altså la hver arbeider benytte en egen løkke som itererer over og løser noen av delproblemene. Sammen løser dermed arbeiderne det tunge hovedproblemet.



Figur 7.2: Forholdet mellom kjøretidene til det parallelle programmet og det sekvensielle. I grafen vises relative kjøretider ved bruk av 1 til 8 tråder

Kapittel 8

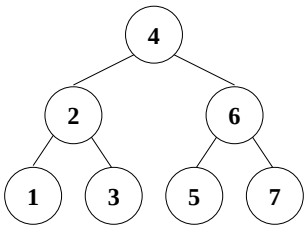
Problem 3: Optimalt søketre - overlappende delproblemer

Dette kapittelet tar for seg ulike implementasjoner av optimalt søketre. Først presenteres den mest vanlige rekursive algoritmen for bruk på tradisjonelle datamaskiner med én prosessor. Deretter sees det på hvordan dette best mulig kan løses ved hjelp av JavaPRP. Til slutt demonstreres et utvalg parallelle algoritmer beregnet på å løse problemet i parallell på multiprosessor.

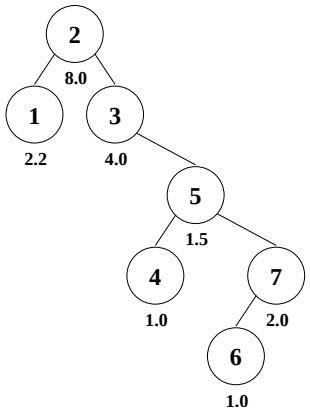
8.1 Binært søketre

Et binært søketre [Wei99a] er en mye brukt datastruktur for å lagre store mengder data. Treet er sortert ved at hver node har en nøkkelverdi. Reglene for sorteringen er at alle noders høyre barn alltid har en nøkkel som er større enn nøkkelen til noden selv, og at nodenes venstre barn har en nøkkel som er mindre. På denne måten er nodene sortert i stigende rekkefølge fra venstre mot høyre. Figur 8.1 viser eksempel på et slikt tre.

Et søk i treet foregår ved å starte i roten og følge grenene nedover. Ved



Figur 8.1: Binært søketre. Tallene er nodenes nøkler.



Figur 8.2: Optimalt binært søketre. Nodene er blitt tildelt en vekt som angir hvor høyt i treet nodene bør ligge.

hver node sammenlignes søkeverdien med nodens nøkkel, og resultatet av sammenligningen bestemmer om man skal fortsette søket mot venstre eller mot høyre.

8.2 Optimalt binært søketre

Tidsforbruket til et søk av denne typen øker proporsjonalt med målnodens dybde i treet. Et søk etter node 1 i treet på figur 8.1 på forrige side tar lenger tid enn et søk etter node 4 fordi noden ligger dypere i treet. Hvis det ofte søkes etter denne noden kan det være en fordel om den ligger høyere. Det kan altså være hensiktsmessig at de mest brukte nodene ligger høyt oppe i treet.

Et optimalt binært søketre [Wei99e] er et binært søketre som er bygd opp etter dette prinsippet. Populære noder ligger høyt oppe nærme roten, mens sjeldent brukte noder ligger dypere.

Før man skal bygge et optimalt søketre må alle noder i treet ha blitt gitt en sannsynlighetsfaktor som forteller hvor ofte det forventes å bli gjort søk etter nøyaktig den noden. Ved å multiplisere denne sannsynlighetsfaktoren med nodens dybde i treet får man en kostnad som forteller hvor mye ressurser man over tid vil bruke på å søke etter denne noden. Summen av disse kostnadene for alle nodene blir den totale kostnaden for hele treet. Et optimalt søketre er det treet som minimerer denne totale kostnadene. Se tabell 8.1 på neste side.

På figur 8.2 har nodene blitt tildelt en sannsynlighetsfaktor. Treet er optimert med hensyn på denne faktoren, slik at de mest populære nodene ligger høyt i treet.

Instans:	Liste av ord w_1, w_2, ..., w_n Sannsynlighet for at de vil bli brukt p_1, p_2, ..., p_n
Problem:	Hva er den optimale oppbygningen av et binært søketre med disse ordene? Med optimal mener vi her den oppbygningen som minimerer forventet total aksesstid. Antall sammenligninger som må til for å aksessere en node ved dybde d er $d + 1$. Vi vil minimere $\sum_{i=1}^{n} p_i(1 + d_i)$.

Tabell 8.1: Problembeskrivelse Optimalt søketre

8.3 Dynamisk programmering

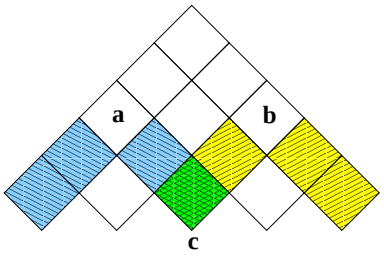
For et optimalt søketre er det slik at treets subtrær alltid også er optimale søketrær. Dette utnyttes når man skal programmere byggingen av et optimalt søketre. Teknikken kalles dynamisk programmering [Wei99c]. Dynamisk programmering går ut på å dele et problem opp i mindre deler som til sammen brukes for å løse det overordnede problemet. Delproblemene deles også på samme måte opp i mindre deler. Svaret fra hvert delproblem lagres underveis i en datastruktur. For optimalt søketre går denne oppdelingen ut på at man beregner optimalt søketre for alle mulige subtrær i treet. Kostnaden til små subtrær brukes som delproblem i større subtrær, og på den måten jobber man seg oppover til man sitter igjen med den optimale kostnaden for hele treet.

I praksis innbærer dette å fylle ut en trekantet tabell som vist på figur 8.3 på neste side. Hvert felt i arrayen tilhører et mulig subtre, og er ment å inneholde kostnaden til dette subtreet. Kostnaden til et subtre er avhengig av kostnaden til andre mindre subtrær som er knyttet til felt lenger ned i tabellen. Det øverste feltet i tabellen skal inneholde kostnaden til hele treet og er avhengig av alle andre felt i tabellen.

8.4 Tradisjonell rekursiv algoritme

Den tradisjonelle algoritmen består i å rekursivt fylle ut en tabell som vist på figur 8.3 på neste side rekursivt. Verdien i hvert enkelt felt i tabellen er lik en funksjon av verdiene som ligger under til venstre og under til høyre for feltet. Det vil si at disse verdiene må beregnes før verdien i det aktuelle feltet kan beregnes. Svaret på selve problemet vil stå i det øverste feltet i tabellen.

Vanligvis løses dette rekursivt med en funksjon som har en referanse til et felt som parameter. Funksjonen henter dataene den trenger for å beregne verdien i feltet ved å kalle seg selv for de nødvendige feltene under.



Figur 8.3: Tabellutfylling ved dynamisk programmering. For å beregne verdien i felt **a** må først verdien i de blå feltene under beregnes. Felt **b** er avhengig av de gule feltene. Felt **c** er delproblem i både **a** og **b**.

Listing 8.1 viser denne algoritmen.

Den trekantede tabellen representeres som en array der kun øverste høyre halvdel av arrayen benyttes. Tabellen blir derfor vridd 45 grader mot høyre slik at det øverste feltet i tabellen tilsvarer felt $[0][n]$ i arrayen der n er antall noder. I listing 8.1 betyr altså $a[i][j]$ kostnaden til subtreet fra node i til j. For et tre med n noder kalles metoden med argumentene $i = 0$ (første node) og $j = n - 1$ (siste node) slik at kostnaden til hele treet returneres.

Listing 8.1: Rekursiv beregning av optimalt søketre

```
1  /* Beregner optimalt søketre for subtre fra node i til node j.
   Kostnaden lagres i a[i][j] */
   public int cost(int i, int j) {
     if (a[i][j] == UBESØKT) {
5      a[i][j] = MAX_VALUE;

       //finn laveste kostnad for alle mulige røtter k
       for (alle k fra i til j) {

10       //beregn kostnad for høyre og venstre subtre, gitt rot k
         int subcost = cost(i, k-1) + cost(k+1, j);

         //sjekk om lavere kostnad er funnet
         if (subcost < a[i][j]) {
15         a[i][j] = sum;
           root[i][j] = k;
         }
       }
       a[i][j] += sigma[i][j]; //sigma er beregnet på forhånd
20   }
     return a[i][j];
   }
```

8.5 Sekvensiell implementasjon

Det er her gjort en implementasjon av algoritmen beskrevet i forrige avsnitt. Programmet bygger altså et optimalt søketre ved hjelp av rekursjon og dynamisk programmering. Resultater etter testkjøringer er vist i tabell 8.2 under.

Problemstørrelse	Tidsforbruk (s)
1500	7.8
4000	226.8

Tabell 8.2: Den sekvensielle algoritmens tidsforbruk for to problemer av ulik størrelse

8.6 Overlappende delproblemer

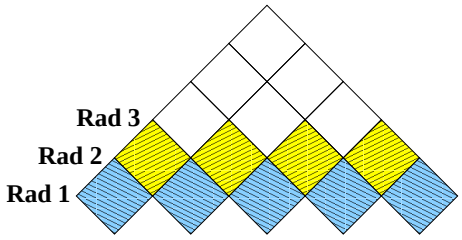
Det spesielle med den rekursive funksjonen beskrevet over er at den vil kalles mange ganger med samme argument. Det skyldes at verdien i et enkelt felt benyttes i utregningene til flere av feltene over i tabellen. Denne egenskapen kan vi kalle overlappende delproblemer.

For å unngå å gjøre samme beregning flere ganger må funksjonen alltid sjekke om den tidligere har blitt kalt med samme argument. I den tradisjonelle algoritmen for optimalt søketre gjøres dette ved å markere besøkte felter slik at verdien i disse returneres umiddelbart uten videre beregninger.

Dette skaper store vanskeligheter når man vil benytte PRP-systemet til å parallellisere denne algoritmen. I PRP forsøker man å dele opp problemet i mange biter ved å bygge opp den øverste delen av rekursjonstreet og håndtere alle subtrærne separat hos ulike arbeidere. Men fordi delproblemene her er overlappende blir det umulig å dele opp problemet i klart adskilte deler. Den vanlige arbeidergenereringsprosessen til PRP vil ikke gi noen ytelsesgevinst fordi arbeidernes subtrær vil være overlappende, slik at alle arbeidere må fylle ut hele nedre del av tabellen.

8.7 Algoritmer for PRP

Til tross for problemet med overlappende delproblemer har optimalt søketre tidligere blitt løst ved hjelp av PRP. Dette ble første gang utprøvd av Victor Eide i 1998 [Eid98]. Jeg har derfor forsøkt å gjenskape dette.



Figur 8.4: Tabellutfylling ved bruk av pumpe. Programmet begynner nederst og tar for seg en og en rad i tabellen. Hele raden fylles ut før man begynner på neste. På figuren fylles først rad 1 (de blå feltene), så rad 2 (de gule feltene) osv.

8.7.1 Eksisterende løsning

Victor Eide skrev i 1998 sin masteroppgave der han presenterer et program skrevet for PRP som bygger et optimalt søketre. Programmet er skrevet i programmeringsspråket C. Ettersom den tradisjonelle algoritmen ikke lar seg parallellisere gjennom PRP på en god måte er det interessant å se hvordan Eide har løst oppgaven.

Løsningen til Eide benytter seg av dynamisk programmering, men i stedet for å benytte rekursjon løper programmet gjennom tabellen iterativt. Programmet tar for seg én og én horisontal rad i tabellen. Første rad er den nederste, og deretter fortsettes det med radene oppover. (se figur 8.4). Dette fungerer fordi alle feltene i en rad kun avhenger av felter på radene under. Når man sørger for å beregne radene i rekkefølge på denne måten sikrer man at delproblemer som et felt er avhengig av allerede er beregnet.

Slik blir det mulig å dele opp hver enkelt rad i mange uavhengige problemer, og disse delproblemene kan beregnes i parallell.

Eides løsning lar administratoren styre denne håndteringen av rader. Administratoren tar for seg en og en rad og fordeler radens delproblemer på arbeiderne. Så venter den på svar fra samtlige arbeidere før den går løs på neste rad. Man kan se på administratoren som en slags **pumpe** som pumper ut problemer til arbeiderne med jevne intervaller.

8.7.2 Ny løsning i nåværende versjon

Det er her forstøkt å gjenskape det overnevnte programmet som et Javaprogram for nåværende versjon av JavaPRP. Men fordi JavaPRP-systemet har endret seg siden den gang har dette imidlertid vist seg ikke å være mulig.

Dette programmet deler tabellen opp i rader som så prosesseres én etter én i en rekursiv metode. Målet var at når programmet skulle kjøres gjennom PRP ville den rekursive metoden parallelliseres. Når en rad var fer-

digberegnet skulle da administratoren pånytt sette i gang parametergenerering for raden over. Dette har vist seg å ikke la seg gjennomføre. Systemet støtter kun at man genererer parametersett en gang, for så å sette i gang med beregninger hos arbeiderne.

Dette tyder på at den utgaven av PRP som Eide benyttet var mer fleksibel enn den nåværende versjonen. Den gamle versjonen av PRP hadde blant annet støtte for flere ekstra nøkkelord som Eide benyttet i denne spesielle løsningen. Av hensyn til brukervennlighet er disse nøkkelordene fjernet i JavaPRP.

En løsning på dette er å modifisere JavaPRP slik at dette igjen mulig. Altså legge til støtte for at administratoren går i en løkke der den genererer nye parametersett og samler opp resultater i hver iterasjon. Det innebærer kanskje at noen nye nøkkelord må innføres på nytt.

8.8 Algoritmer for multiprosessor

Da optimalt søketre ikke lot seg løse gjennom JavaPRP er det interessant å se om det kan være lettere å implementere dette i parallell på multiprosessor. Her vil det presenteres ulike algoritmer for å løse dette. Algoritmene benytter ulike strategier for å håndtere de overlappende delproblemene.

Med unntak av Topdown-algoritmen er det foretatt tester med probleminstanser av størrelse 1500 og 4000. Antall tråder har variert fra 1 til 12. Alle tester har blitt utført to ganger, der det dårligste resultatet er forkastet. Tabellene viser utdrag av testresultatene.

8.8.1 Pumpe

Eides PRP-løsning inspirerte meg til å implementere et program for flerkjerneprosessor som benytter samme teknikk for å parallellisere optimalt søketre. I likhet med Eides program tar dette programmet i bruk en slags pumpe som tar for seg en og en rad i tabellen. For hver rad pumpes det ut delproblemer som løses i parallell. Når delproblemene er løst samles resultatene inn og nye delproblemer pumpes ut for neste rad.

Programmet benytter seg av en administratorkomponent samt et bestemt antall tråder, hver knyttet til en instans av en arbeider-klasse. Administratoren forteller arbeiderene hvilke oppgaver de skal løse og setter dem i gang. Så venter den til alle arbeiderne er ferdig og samler inn resultatene. Deretter kan den fortsette med å pumpe ut flere oppgaver.

Dette programmet benytter seg ikke av felles hukommelse. Derfor behøver man ikke å tenke på hvordan det skal håndteres. Trådene deler i stedet data ved at administratoren henter resultater fra arbeiderne. Administratoren fyller ut delproblemtabellen, og sender data til arbeiderne etter hvert.

Antall tråder	Problemstørrelse	Tidsforbruk (s)
1	1500	12.0
2	1500	9.3
4	1500	9.7
8	1500	10.3
12	1500	9.5
1	4000	273.2
2	4000	164.0
4	4000	135.3
8	4000	148.9
12	4000	154.4

Tabell 8.3: Pumpealgoritmens tidsforbruk ved bruk av forskjellig antall tråder på CPU med fire kjerner

Det er foretatt testkjøringer med probleminstanser av størrelse 1500 og 4000. Resultatene er vist i tabell 8.3.

For problemet med 1500 noder gir programmet ingen ytelsesforbedring ved økning av antall tråder. Dette problemet er altså for lite til at parallelliseringen gir noen gevinst i dette programmet. Ved bruk av 4000 noder gir imidlertid programmet bedre resultater. Ytelsen til programmet er omtrent dobbelt så høy ved bruk av 4 tråder som ved bruk av 1 tråd. Legg merke til at når man øker antall tråder ytterligere, til for eksempel 8, så går ytelsen ned. Det kan tyde på at programmet bruker mye tid på å synkronisere arbeiderne med administratoren og å sende data mellom disse.

8.8.2 Top-down

De overlappende delproblemene gjorde at den tradisjonelle rekursive algoritmen viste seg å være vanskelig å parallellisere ved hjelp av PRP. Jeg har derfor forsøkt å finne ut om det likevel er mulig å kjøre denne algoritmen i parallell på en multikjerneprosessor.

Fordi alle de ulike instansene av den rekursive metoden jobber mot en felles datastruktur (en tabell) må programmet benytte seg av en form for synkronisering. Man må beskytte de felles dataene slik at ikke flere arbeidere kan skrive til samme data samtidig.

Jeg har tatt utgangspunkt i å kjøre den rekursive metoden i parallell på vanlig måte. Altså ved først å generere arbeidere, så eksekvere arbeiderne i parallell og til slutt samle opp resultater.

Det ble først forsøkt å eksekvere dette uten bruk av noen form for synkronisering. Det imidlertid ville ikke fungere fordi arbeiderne har overlappende delproblemer og samtidig forsøkte å løse de samme problemene. Dermed forsøkte arbeiderene å samtidig beregne de samme feltene i tabellen. Man trenger en form for synkronisering slik at kun en arbeider jobber

Antall tråder	Problemstørrelse	Tidsforbruk (s)
1	400	3.7
2	400	112.4
4	400	40.8
8	400	96.4
12	400	77.8

Tabell 8.4: Top-down-algoritmens tidsforbruk ved bruk av forskjellig antall tråder på CPU med fire kjerner

på ett felt om gangen. I praksis vil det si at det til en hver tid bare er en arbeider som skal få lov til å kalle den rekursive metoden med et bestemt parametersett.

Jeg har løst dette ved å legge til en synkronisert metode som setter et bit i en array knyttet til parametersettet. Metoden kalles på starten av den rekursive metoden. I listing 8.2 er metoden representert ved kallet `aquire_lock()`. Når en annen arbeider kaller metoden med samme parametersett vil den finne ut at låsen er låst og må vente til den frigjøres.

Listing 8.2: Parallell topdown-løsning av optimalt søketre

```
1  public int cost(int i, int j) {
     boolean lock = LOCKED;
     //spinn inntil låsen er åpen og kan låses
     while (lock == LOCKED) {
5      lock = aquire_lock(i, j);
     }
     //løs optimalt søketre for subtre {i, j}
     a[i][j] = ...

10   release_lock(i, j);
     return a[i][j];
   }
```

Programmet fungerer og kommer frem til riktig resultat, men kjøretiden er dårlig. Som vist i tabell 8.4 kan ikke programmet håndtere svært store problemer så det har bare blitt forsøkt med probleminstanser opp til størrelse 400. Det går klart raskest når bare en tråd benyttes. Ved bruk av flere tråder bruker programmet svært mye tid på synkronisering. Resultatene varierer mye, og de er tilsynelatende tilfeldige.

8.8.3 Barrier med vanlig array

Neste program som demonstreres benytter seg av barriersynkronsiering. Dette fungerer ved det at opprettes en barrier (en vegg) som alle trådene av og til møter (de gjør et kall på `barrier.wait()`). Når trådene møter en slik vegg må de vente inntil tilstrekkelig mange tråder (ofte alle) møter den samme veggen. Først da får de slippe videre i sin eksekvering.[Tan01]

I dette programmet får trådene utdelt absolutt alle problemene de skal løse før eksekvering, altså ikke bare de som er på ett nivå. Deretter starter alle tråder å løse sine problemer, først på laveste nivå og så jobber de seg oppover. Mellom hvert nivå møter alle trådene en barrier, der de må vente på at alle andre tråder er ferdig med nivået før de kan fortsette. På denne måten sikrer man at delproblemene løses i riktig rekkefølge. Man sørger for at før en tråd løser et delproblem på et visst nivå må alle delproblemene på nivåene under være løst. Listing 8.3 viser denne algoritmen.

Listing 8.3: Arbeidernes metode for problemløsing

```
1  public void run() {
     for (int level = 1; level < n; level++) {
       // løs mine oppgaver på dette nivået
       int i = 0;
5      for (int j = level; j < n; j++) {

         if (j \% nThreads == id) {
           solve(i, j);
         }
10       i++;
       }
       //vent på andre arbeidere
       barrier.await();
     }
15 }
```

Felles datastruktur

I dette programmet jobber alle trådene mot en felles datastruktur. De både leser og skriver fra denne, og man kan risikere at flere tråder vil lese fra samme data i parallell. Her er det altså viktig å ta hensyn til race conditions og asynkron cache.

I det sekvensielle programmet ble det brukt en todimensjonal array slik som den på figur 8.3 på side 48. For å finne ut om det kan oppstå problemer ved å bruke en tilsvarende array i det parallelle programmet vil jeg se på hvordan trådene leser til og skriver fra arrayen.

Mellom hver iterasjon, altså på hver rad i tabellen, har en tråd ansvar for å løse sine utvalgte delproblemer. Svaret på disse skrives inn i tabellen i de tilhørende feltene. Tråden er nå den eneste som skriver til disse feltene, så vi får her ingen race conditions. For å finne svaret på delproblemene benytter den seg av data på rader lenger ned i tabellen. Disse dataene kan benyttes og leses av flere tråder i parallell, og er regnet ut tidligere da trådene jobbet på de lavere nivåene.

Man skulle altså tro at man kan få problemer med asynkron cache i dette programmet. En tråd benytter seg jo av data som andre tråder tidligere har skrevet i arrayen.

Antall tråder	Problemstørrelse	Tidsforbruk (s)
1	1500	7.6
2	1500	4.1
4	1500	2.1
8	1500	1.4
12	1500	2.6
1	4000	228.8
2	4000	146.5
4	4000	90.1
8	4000	59.7
12	4000	110.0

Tabell 8.5: Barrier med vanlig array. Algoritmens tidsforbruk ved bruk av forskjellig antall tråder på CPU med fire kjerner

Her hjelper imidlertid barriersynkroniseringen oss. Synkroniseringen mellom hvert nivå fører til at alle tråder synkroniserer sin cache med hovedlageret. Dermed blir de data som en tråd har skrevet på det aktuelle nivået flushet ut i hovedlageret slik at andre tråder trygt kan lese de når de jobber på høyere nivåer.

Barriersynkroniseringen har altså to nytteverdier i dette programmet:

1. Den sørger for at trådene kun jobber sammen på ett nivå om gangen.

2. Den sørger for at trådene flusher sin cache mellom hvert nivå slik at andre tråder trygt kan lese dataene de nettopp har skrevet.

Resultatet etter testkjøringer vises i tabell 8.5. Man kan se at programmet skalerer fint med antall tråder, både for problemet med 1500 noder og for problemet med 4000 noder. Programmet yter best ved bruk av 8 tråder, til tross for at prosessoren kun har 4 kjerner. Dette betyr at prosessorens multithreadingegenskaper bidrar til å effektivisere algoritmen. Ved bruk av 1 tråd yter programmet like godt som det sekvensielle. Ved bruk av 8 tråder yter det nesten 4 ganger bedre.

8.8.4 Barrier med AtomicIntegerArray

Det kunne imidlertid hende at trådene skulle behøve å aksessere en felles array på en slik måte at det faktisk oppstår problemer. Da finnes det flere måter å håndtere dette på.

Man kan benytte synkronisering ved å ha synkroniserte metoder for lesing fra og skriving til arrayen. Men det ville føre til at man låser hele arrayen. Denne låsen ville blitt låst hver enkelt aksess, noe som ikke gir trådene mulighet til å aksessere ulike deler av arrayen samtidig.

Antall tråder	Problemstørrelse	Tidsforbruk (s)
1	1500	13.4
2	1500	7.2
4	1500	4.9
8	1500	2.3
12	1500	4.4
1	4000	365.7
2	4000	199.5
4	4000	108.6
8	4000	78.3
12	4000	129.9

Tabell 8.6: Barrier med AtomicIntegerArray. Algoritmens tidsforbruk ved bruk av forskjellig antall tråder på CPU med fire kjerner

En annen mulighet er å benytte Javas innebygde pakke for atomiske datatyper. I denne pakken ligger en klasse som heter `AtomicIntegerArray`. Klassen representerer en array som kan aksesseres av flere tråder samtidig fordi alle operasjoner blir gjort atomisk.

Selv om en vanlig todimensjonal array viste seg å fungere problemfritt vil jeg også lage en variant av programmet som benytter `AtomicIntegerArray`. Dette programmet kan sees på som et eksempel på et mer generelt program der det tillates at flere tråder fritt kan lese fra og skrive til samme datastruktur.

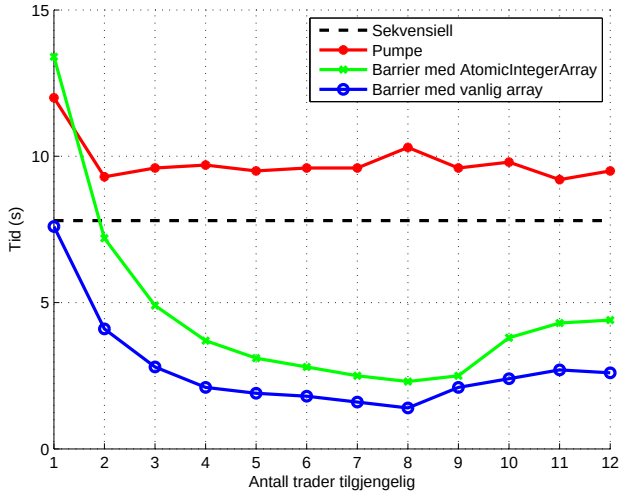
Fordi vi her behøver en todimensjonal array benytter jeg en array av instanser av denne klassen (`AtomicIntegerArray[]`). De mange trådene aksesserer arrayen ved hjelp av klassens get- og set-metoder.

Resultatet av testkjøringene er vist i tabell 8.6. Også dette programmet skalerer fint med antall tråder. Det yter imidlertid ikke like godt som programmet som benytter vanlig array. Ved bruk av 8 tråder bruker dette programmet omtrent 30% lenger tid enn 'Barrier med array'.

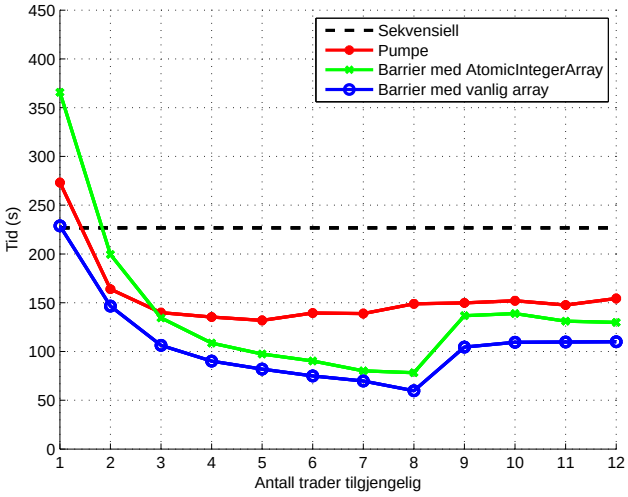
8.9 Sammenligning

Figur 8.5 på neste side og 8.6 på neste side viser sammenligninger av resultatene fra programmene 'Pumpe', 'Barrier med vanlig array', 'Barrier med AtomicIntegerArray' og 'Sekvensiell'.

For problemet med 1500 noder har 'Pumpe' dårligere ytelse enn det sekvensielle programmet. 'Barrier med vanlig array' har mye bedre ytelse enn det sekvensielle programmet ved bruk av mange tråder. Ved bruk av 1 tråd yter 'Barrier med vanlig array' like godt som det sekvensielle. 'Barrie med AtomicIntegerArray' yter også godt ved bruk av mange tråder, men ikke like godt som 'Barrier med vanlig array'. Ved bruk av 1 eller 2 tråder



Figur 8.5: Kjøretider for optimalt søketre med 1500 noder



Figur 8.6: Kjøretider for optimalt søketre med 4000 noder

yter det dårligere enn det sekvensielle.

For problemet med 4000 noder er også 'Pumpe' raskere enn det sekvensielle. Men det er ikke like raskt og skalerer ikke like godt som de andre programmene. 'Barrier med AtomicIntegerArray' er raskt ved bruk av mange tråder, men blir så vidt slått av 'Barrier med vanlig array'.

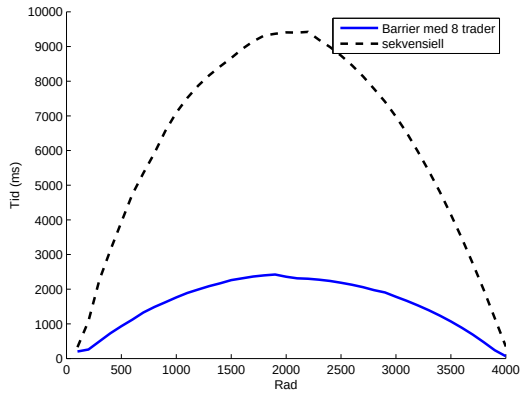
8.10 Eksperiment: Blanding av sekvensiell og parallell løsning

Da Eide skulle løse optimalt søketre ved hjelp av JavaPRP fant han ut at man dro størst fordel av å parallellisere de midterste delproblemene, altså de delproblemene som var knyttet til radene langs midten av tabellen. Delproblemene i radene øverst og nederst burde heller beregnes sekvensielt.

Det kommer av at denne algoritmen er mest beregningstung langs midten av tabellen. På de nederste radene er det mange delproblemer, men hvert av disse kan til gjengjeld beregnes svært raskt. På de øverste radene i tabellen er delproblemene mer beregningstunge, men det er til gjengjeld svært få av de. Dermed blir disse radene vanskeligere å parallellisere fordi det er begrenset hvor mange delproblemer man kan dele radene inn i.

For å finne ut om dette også gjelder for parallellisering på multiprosessor har det blitt implementert et program som måler hvor lang tid det parallelle programmet bruker på hver rad. Dette sammenlignes så med en måling av hvor lang tid det sekvensielle programmet bruker på hver rad. Programmet er basert på programmet som benytter seg av barrier og vanlig array.

Resultatet vises på figur 8.7. Figuren viser at det stemmer at problemet er mest beregningstungt langs midten av tabellen. Man kan også se at kurven til det parallelle programmet ligger under kurven til det sekvensielle på alle nivåer. Det gjelder også for de første og de siste nivåene. Med andre ord er det ingen grunn til å benytte sekvensiell algoritme på deler av problemet hvis man parallelliserer på multiprosessor. Overhead knyttet til selve parallelliseringen er for liten.



Figur 8.7: Kjøretid for ulike nivåer. Figuren viser kjøretiden til 'Barrier' sammenlignet med kjøretiden til det sekvensielle programmet for ulike nivåer i tabellutfyllingen

8.11 Oppsummering - gjentatt parallelliserbar løkke

Dette kapittelet har tatt for seg parallelisering av optimalt søketre. Det er et problem med overlappende delproblemer. Algoritmen for dette viste seg å være vanskelig å parallellisere direkte, men den lot seg bygge om til en iterativ algoritme med en liten parallelliserbar del som ble gjentatt mange ganger. Man kunne i dette tilfelle altså ikke parallellisere hele algoritmen i sin helhet, men man kunne utføre en enkeltstående parallellisering hver gang algoritmen går inn i den parallelliserbare delen.

Generelt sett består denne algoritmen av en dobbel løkke der den ytre løkka må eksekveres sekvensielt, mens den indre løkka kan parallelliseres. Listing 8.4 viser eksempel på dette.

Listing 8.4: Gjentatt parallelliserbar løkke

```
1  void løsProblem() {
     //ingen parallellisering!
     for( hver gruppe delproblemer som må løses sekvensielt ) {
       løsDelproblemer( ... );
5    }
   }

   int løsDelproblemer( ... ) {

10   //parallelliserbar løkke
     for( hvert delproblem som kan løses i parallel ) {

     }
     return svar;
15 }
```

Den parallelliserbare løkken kan parallelliseres på samme måte som løkken vist kapittelet om Goldbach ved å fordele delproblemer på en mengde arbeidere.

Den raskeste implementasjonen av denne algoritmen benytter seg av barriersynkronisering. Det fungerer ved at alle trådene får tildelt alle sine delproblemer, og synkroniseres underveis ved hjelp av barrier slik at delproblemene løses i riktig rekkefølge i henhold til den ytre løkka.

Kapittel 9

Automatisk parallellisering

De tre foregående kapitlene har beskrevet manuell parallellisering på multiprosessor. Når vi nå går over på automatisk parallellisering finnes det tre muligheter: (1) Vi kan benytte det eksisterende systemet JavaPRP. Dette systemet har støtte for multiprosessorer, men dette gir noen begrensninger. (2) Vi kan videreutvikle JavaPRP for å gi systemet bedre støtte for multiprosessor. (3) Vi kan lage en ny preprosessor som er skreddersydd for multiprosessor. Denne oppgaven ta for seg det sistnevnte, og dette kapittelet vil begrunne hvorfor.

9.1 JavaPRP på multiprosessor

Selv om det eksisterende systemet JavaPRP fokuserer på å parallellisere ved bruk av mange maskiner over nettverk har det nylig også blitt implementert støtte for parallellisering på multiprosessor [Syv07]. Dersom en av maskinene i nettverket er av en slik type vil JavaPRP sørge for at multiprosessoren får mulighet til å løse flere oppgaver i parallell. Dette fungerer ved at arbeiderprogrammene, altså programmene som kjører på de ulike maskinene, sjekker hvor mange prosessorer de har tilgjengelig. På bakgrunn av det setter den i gang flere uavhengige arbeidere.

Dette kan man benytte seg av også hvis man bare skal parallellisere på en enkelt multiprosessor. Ved å starte et arbeiderprogram lokalt på prosessoren vil det programmet starte mange arbeidere som vil kunne løse problemet i parallell.

Det er imidlertid noen ulemper med denne fremgangsmåten. Fordi den automatiske parallelliseringen helst skal fungere så enkelt som mulig har vi stor fokus på brukervennlighet. Når vi kun trenger å forholde oss til en maskin bør vi bare behøve å forholde oss til ett program under parallelliseringen. Men ved bruk av JavaPRP er man nødt til å starte opp både en

administrator og en arbeider. Deretter må det parallelle systemet startes opp via administratoren.

En annen ulempe ved å benytte JavaPRP til parallellisering på multiprosessor er at man ikke kan dra noen fordel av felles hukommelse, da JavaPRP ikke er bygd for å takle dette. Dette eliminerer effektiv parallellisering av problemer som for eksempel Travelling Salesperson, der bruk av en felles cutoffvariabel gir stor ytelsesforbedring.

Dessuten er JavaPRP uegnet til parallellisering av løkker. For å parallellisere løkker ved hjelp av JavaPRP må man først skrive om algoritmen slik at den blir rekursiv. Det er en tidkrevende prosess ,men det fungerer for enkle løkke slik som i Goldbach. Pumpeproblemer støttes derimot ikke. Slike problemer er det ikke mulig å løse med den nåværende versjonen av JavaPRP.

9.2 Videreutvikling av JavaPRP

På sikt vil det være en mulighet å videreutvikle JavaPRP til å støtte disse tingene. Hvis man benytter denne fremgangsmåten vil det imidlertid være vanskelig å gjøre noe brukervennlighetsaspektet. JavaPRP er et tungvindt system å bruke dersom man bare skal parallellisere på en multiprosessor.

9.3 Ny preprosessor

Det neste kapittelet vil ta for seg det tredje alternativet. Det vil bli presentert et system som er skreddersydd for parallellisering på multiprosessor. Systemet heter MPRP (Multicore PRP).

Målet med MPRP er å kunne forenkles parallelliseringsprosessen i scenarioer der man bare skal parallellisere på en maskin. På et slikt system er det ikke være nødvendig å skille ut administratoren og arbeideren i separate komponenter/programmer. Det ideelle er at resultatet etter preprosessering er et enkelt program som i seg selv inneholder alle disse tingene. Etter preprosessering bør man helt enkelt kunne starte opp dette programmet, så utfører det problemløsingen i parallell automatisk.

I dette nye systemet er det bygd inn støtte for at de ulike arbeiderne kan dele hukommelse og på den måten lese og skrive til samme data.

Et annet system som minner om MPRP er OpenMP [Ope08]. OpenMP (Open Multi-Processing) er et programmeringsgrensesnitt (API) som støtter parallellprogrammering med delt hukommelse i C/C++ og Fortran. Systemet fungerer ved at en støttet kompilator leser annoteringer i brukerens kode for å generere parallelle programmer. MPRP skiller seg fra OpenMP ved at det støtter andre former for parallellisering (for eksempel parallellisering av rekursjon). En annen forskjell er at det er laget for Java og at det ikke krever en spesiell kompilator for å fungere.

Kapittel 10

Presentasjon av MPRP

Basert på det som hittil er skrevet i denne oppgaven om parallellisering på multiprosessor vil vi nå gå løs på automatiseringen av dette. Dette kapittelet vil presentere det nye systemet MPRP.

10.1 Tre programmeringsteknikker

I likhet med JavaPRP har MPRP støtte for automatisk parallellisering av rekursjon. Dermed er det i stand til å parallellisere for eksempel Travelling Salesperson (kapittel 6 på side 33). I tillegg kan systemet parallellisere problemene Goldbach (kapittel 7 på side 39) og Optimalt søketre (kapittel 8 på side 45). Det har med andre ord støtte for problemer der det benyttes løkke, og problemer med gjentatt parallelliserbar løkke.

De tre programmeringsteknikkene som støttes kunne implementeres på ulike måter. Man kunne enten benytte en felles mekanisme som støtter alle tre eller man kunne håndtere de tre teknikkene forskjellig.

Det er trolig mulig å implementere alt dette som en felles mekanisme. Blant annet så er det slik at enhver løkke kan skrives om til å bli en rekursiv funksjon. Det er imidlertid trolig en unødvendig tung prosess når målet er å oppnå så raske parallelle programmer som mulig.

Her har det imidlertid blitt valgt å implementere tre ulike mekanismer for de tre programmeringsteknikkene. Preprosessoren går altså inn i tre forskjellige moduser avhengig av hvilken teknikk som er benyttet. Dette åpner for optimaliseringer av de spesifikke modulene.

For rekursivitet benytter preprosessoren de samme teknikkene som ble brukt i det parallelle programmet for Traveling Salesperson. Dette er altså sterkt inspirert av JavaPRP.

Løkkehåndteringen minner om virkemåten til det parallelle programmet for Goldbach. Ettersom løkker ennå ikke er støttet av JavaPRP har

Figur 10.1: Modell av virkemåten til preprosessoren i MPRP. Systemet mates med et sekvensielt program og genererer et parallelt program basert på det sekvensielle.

prinsippene for hvordan disse skal parallelliseres blitt utviklet fra bunn av. Kanskje vil dette prosjektet ligge til grunn for implementasjon av dette i JavaPRP.

Videre har systemet fått støtte for gjentatt parallell løkke. Dette er heller ikke støtte av JavaPRP så denne funksjonaliteten er inspirert av metodene benyttet i de parallelle programmene for optimalt søketre.

10.2 Bruk av systemet

Det stilles høye krav til brukervennlighet. Alt brukeren skal behøve å gjøre er å markere parallelliserbar kode i programmet med noen enkle kommentarer. Disse kodeordene gjør at preprosessoren forstår hvordan den skal forandre på programmet. En viktig forskjell fra JavaPRP er at det her ikke er nødvendig å måtte kjøre resultatet ved hjelp av et administratorprogram og arbeiderprogrammer. I MPRP blir alle disse komponentene bakt inn i resultatprogrammet slik at den ene resultatfilen er alt man trenger. Målet er at resultatklassen sett utenfra skal ha nøyaktig samme funksjonalitet som den opprinnelige klassen.

Figur 10.1 viser bruk av systemet. MPRP er i sin helhet en preprosessor som mates med et program laget av en bruker. Resultatet blir et nytt program som så godt som mulig har samme funksjonalitet som det opprinnelige. Forskjellen er at resultatprogrammet utfører beregningene i parallell.

10.2.1 Kodeord

Utgangspunktet før preprosessering skal være en klasse skrevet i Java der det er ønskelig at deler av koden skal parallelliseres. Koden det gjelder må være en rekursiv funksjon eller en løkke. For at preprosessoren skal forstå hvilken kode det gjelder må det legges inn noen kodeord i form av kommentarer på faste steder. Det benyttes ulike kodeord avhengig av hva slags programmeringsteknikk som er brukt i den parallelliserbare koden.

Rekursjon

Dersom den parallelliserbare koden er en rekursiv metode skal koden markers på samme måte som i JavaPRP. Som i JavaPRP skal derfor følgende to kodeord benyttes:

- `/* PRP_PROC */`
  markerer den rekursive metoden. Kommentaren settes inn på linjen over metodens header.

- `/* PRP_CALL */`
  markerer det rekursive kallet. Kommentaren settes inn på linjen over utrykket som inneholder kallet.

Løkke

Løkken som det er ønskelig å parallellisere plasseres i en egen metode. Inne i løkken må det være et kall på en annen metode der selve beregningene utføres. Det må benyttes to kodeord:

- `/* PRP_LOOP */`
  markerer metoden som inneholder løkken.

- `/* PRP_CALL */`
  markerer et metodekall inne i løkken.

Bortsett fra selve løkken og løkkens kall bør denne metoden inneholde så lite kode som mulig. Det kommer av at all kode i denne metoden blir kjørt av alle arbeidere, med unntak av selve metodekallet inne i løkken. Arbeiderne veksler på å utføre metodekallet.

Gjentatt parallelliserbar løkke

I noen programmer er det slik at den parallelliserbare løkken eksekveres gjentatte ganger. Det skjer hvis den parallelliserbare metoden kalles flere ganger fra en løkke i en annen metode. Dette er tilfellet i for eksempel algoritmen for optimalt søketre. Vi kaller denne metoden for en **pumpe**. For å effektivisere eksekveringen bør også denne metoden annoteres med kodeord. Følgende to kodeord benyttes:

- `/* PUMP_LOOP */`
  markerer pumpemetoden

- `/* PUMP_CALL */`
  markerer kallet på den parallelliserbare metoden

10.3 Felles hukommelse

Parallelle programmer generert av MPRP kan dra nytte av den felles hukommelsen man har tilgang til på en multiprosessor. Her har brukeren av systemet stor frihet til å avgjøre hvordan dette skal implementeres. Brukeren kan benytte felles hukommelse ved å legg inn kall på globale metoder og datastrukturer inne i den parallelliserbare metoden. I det genererte parallelle programmet vil disse kallene utføres i parallell av arbeiderne. Dersom man benytter seg av dette er det derfor viktig at man tar hensyn til problemene som kan oppstå ved bruk av felles hukommelse. For eksempel kan det være nødvendig at felles skrivemetoder blir benyttet deklareres som synkronisert. Dette er beskrevet i seksjon 5.4 på side 26.

Listing 10.1 viser eksempel på bruk av felles hukommelse. Listingen viser en rekursiv metode før preprosessering der det er lagt inn et kall på en felles metode som brukes til å øke verdien til en felles variabel. I det preprosesserte programmet vil dette kallet utføres i parallell av alle arbeiderne. Metoden er derfor deklarert som `synchronized` for at de trygt skal kunne kalle den i parallell.

Listing 10.1: Kall på felles metode

```
1 int counter = 0;

  synchronized void increaseCounter() {
    counter++;
5 }


  /* PRP_PROC */
  void solveProblem() {
10   ...

    increaseCounter();
    ...
  }
```

10.4 Parallellisering av rekursjon

MPRP håndterer rekursive metoder på en måte som ligner på den som er brukt i JavaPRP. Til forskjell fra JavaPRP produserer MPRP bare en fil. Denne ene filen blir et fullverdig program som inneholder alle mekanismene som skal til for parallellisering.

Den nye klassen som produseres har skal så godt som mulig nøyaktig samme funksjonalitet som den opprinnelige, med unntak av at deler av koden utføres i parallell.

På samme måte som JavaPRP benytter MPRP seg av en administrator/arbeidermodell. Parametersett og returverdier håndteres også på en lignende måte.

I JavaPRP representeres disse ulike komponentene som ulike klasser som kjører på ulike maskiner. Denne gangen blir derimot alt implementert som metoder og indre klasser i resultatprogrammet.

De tre sentrale komponentene er implementert som følger:

- Administrator
  I JavaPRP er dette en ny klasse som kjører på administratormaskinen. I MPRP er dette en metode i den genererte koden. Metoden som benyttes er metoden som opprinnelig var rekursiv. Et kall på denne metoden blir dermed et kall på administratoren.

- Arbeider
  I JavaPRP er dette et program som kjøres på alle arbeiderne. I MPRP er dette implementert som en indre klasse i den genererte koden.

- Parametersett og returverdier
  I JavaPRP er dette objekter som distribueres over nettet. MPRP implementerer disse som objektvariabler inne i arbeideren. Dette beskrives nærmere i seksjon 10.4.2 på neste side.

10.4.1 Administratoren

Administratoren tar seg av håndteringen av alle arbeiderne. Administratoren er det første som bør kalles ved eksekvering. Den skal fungere som en sentral styringsenhet som fordeler oppgaver og samler inn resultater. Til slutt er det administratoren som kommer frem til det endelige svaret.

Som nevnt tidligere er det viktig at resultatprogrammet har mest mulig lik funksjonalitet som det opprinnelige. Det innebærer at de opprinnelige metodene bør utføre det samme etter preprosessering som før preprosessering. Dette gjelder også den rekursive metoden brukeren ønsker å parallellisere. Et kall på denne metoden bør sette i gang beregningene. Derfor bør et kall på metoden etter preprosessering sette i gang de parallelle beregningene. Dette burde derfor medføre et kall på administratoren.

I MPRP løses dette ved at administratoren implementeres direkte i metoden som opprinnelig var den rekursive. All koden i metoden blir altså byttet ut med administratorkode. Den opprinnelige rekursive koden blir i stedet plassert andre steder.

På denne måten vil alle kall på den opprinnelige rekursive metoden bli byttet ut med kall på administratoren. Administratoren setter i gang parallelle arbeidere og kommer frem til et felles svar. Administratoren vil så returnere dette svaret, på samme måte som den rekursive metoden returnerer sitt endelige svar. Listing 10.2 på neste side og listing 10.3 på neste side viser denne forskjellen. De to metodene utfører samme oppgave, men på to forskjellige måter.

Listing 10.2: Rekursiv metode før preprosessering

```
1  /* PRP_PROC */
   int solveProblem( ... ) {
     //gjøre beregninger rekursivt
     solveProblem( ... );
5
     return svar;
   }
```

Listing 10.3: Rekursiv metode etter preprosessering

```
1  /* PRP_PROC */
   int solveProblem( ... ) {
     //Administrator
     //Opprett arbeidere
5    //Fordel oppgaver og sett i gang
     //samle resultater

     return svar;
   }
```

10.4.2 Arbeideren

Arbeideren vil være den delen av systemet som utfører selve beregningene. En arbeider skal være i stand til å løse et delproblem, slik at mange arbeidere sammen løser hele problemet. I JavaPRP blir arbeiderne representert som selvstendige programmer. Det kjøres en arbeider på hver maskin. I MPRP produseres kun ett program, og arbeiderne implementeres derfor som en indre klasse i dette. Denne klassen kalles `PRPWorker`. Administratoren oppretter en instans av `PRPWorker` hver gang den trenger en arbeider. Arbeiderne kjører i hver sin tråd og er derfor i stand til å kjøre i parallell.

Arbeideren inneholder tre viktige metoder; `solve()`, `createChildren()` og `gatherResult()`. Alle disse metodene er modifiserte utgaver av den opprinnelige rekursive metoden og blir benyttet under ulike faser av eksekveringen:

- `solve()` er en fullstendig kopi av den opprinnelige rekursive metoden. Denne metoden kalles når en arbeider skal løse sitt delproblem.
- `createChildren()` (opprett barn) benyttes under arbeidergenerering.
- `gatherResults()` (samle resultater) benyttes til slutt av administratoren under innsamling av resultater.

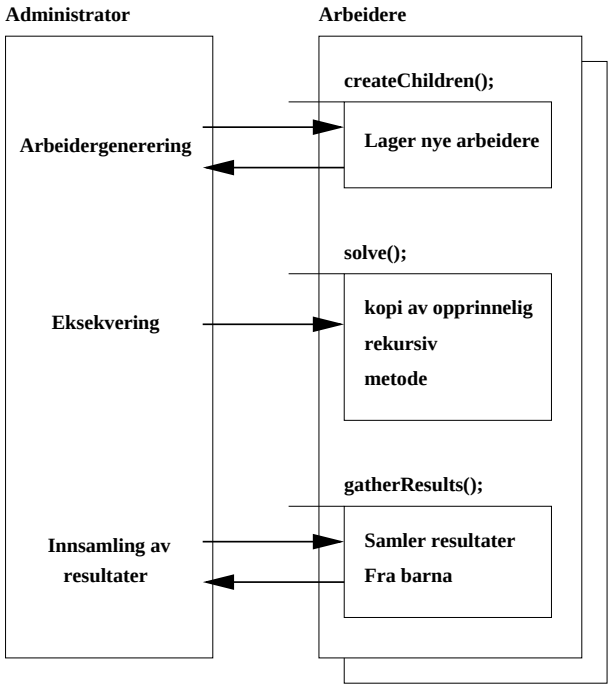
Arbeideren inneholder også parametersett og resultatverdier. I JavaPRP ble disse representert som egne klasser som ble sendt frem og tilbake over

nettverket. I MPRP er dette innbakt i arbeideren. Ved opprettelse av en arbeider sendes det altså med hvilke parametere den skal jobbe mot. Når en arbeider har kommet frem til et resultat lagrer den dette lokalt.

På denne måten trenger administratoren kun å forholde seg til arbeidere. Opprettelse av parametre skjer gjennom opprettelse av arbeidere, og innsamling av resultater skjer gjennom kall på arbeidere.

10.4.3 Tre faser

Utførelsen av beregningene skjer gjennom tre faser. Dette er de samme tre fasene som benyttes i JavaPRP. Figur 10.2 viser hvordan administratoren går gjennom de tre fasene.

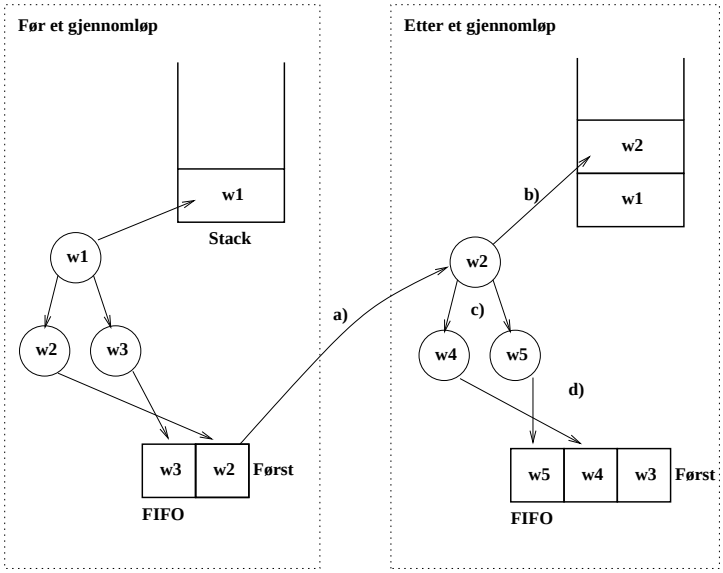


Figur 10.2: Administratorens tre faser under eksekvering.

Først utføres arbeidergenerering. Dette tilsvarer parametergenereringen i JavaPRP, men fordi parmeterene er innbakt i arbeiderne utføres dette som arbeidergenerering. Deretter kommer eksekvering, der arbeiderne utfører beregning av sitt delproblem, og til slutt foretas innsamling av resultater.

10.4.4 Arbeidergenerering

Etter at administratoren har sjekket hvor mange prosessorer som er tilgjengelig starter den med arbeidergenerering. Genereringen fungerer på samme måte som i JavaPRP ved hjelp av to datastrukturer. En FIFO-kø og en stack. Figur 10.3 viser hvordan dette fungerer.



Figur 10.3: Figuren viser ett gjennomløp av løkken som genererer arbeidere. w1, w2 osv. representerer arbeidere (workers). Løkken utfører fire skritt som legger arbeidere på stacken og FIFO-køen.

Administratoren går i en løkke der den utfører følgende skritt:

1. Popper første arbeider fra køen (skritt a)
2. Pusher denne på stacken (skritt b)
3. Genererer nye parametersett ved å ta utgangspunkt i arbeideren som akkurat er hentet ut(skritt c). Dette gjøres ved å kalle metoden `createChildren()` i arbeideren. Denne metoden kjører den opprinnelige rekursive koden frem til kallstedet der kallet er byttet ut med konstruksjon av nye arbeidere. Pekere til barna lagres i arbeideren i en liste.
4. Legger de nye arbeiderne som ble generert i listen (skritt d)

På denne måten jobber administratoren seg nedover i rekursjonstreet mens den genererer nye arbeidere for hvert delproblem den støter på. Til slutt inneholder FIFO-køen mange nok arbeidere til at kodeeksekveringen kan starte.

10.4.5 Kodeeksekvering

Etter arbeidergenerering ligger arbeiderne klare i FIFO-køen. Det er generert mange flere arbeidere enn antall prosessorer tilgjengelig, slik at man unngår at noen oppgaver blir for store. Ved å la Arbeiderklassen implementere interfacet `Runnable` kan instansene tilordnes hver sin tråd og eksekveres i parallell. En måte å gjøre dette på er å bruke Javas innebygde klasse `ExecutorService`.

Hver arbeider utfører sine beregninger i hele sitt rekursive subtre og legger resultatet i en objektvariabel `result`. Denne variabelen må være av samme type som returverdien til den rekursive metoden. Så i stedet for å returnere en verdi blir denne verdien lagt i denne variabelen.

10.4.6 Svargenerering

Generering av sluttresultat foregår på samme måte som i JavaPRP ved at en løkke tar for seg en og en arbeider på stacken. For hver arbeider genereres et resultat ved å samle den inn resultatet fra arbeiderens barn. Arbeiderne på stacken er som kjent arbeidere som ennå ikke har beregnet noe resultat men som derimot har en liste med barn som har regnet ut et midlertidig resultat og lagret dette som en objektvariabel. Innsamlingen av disse resultatene skjer i metoden `gatherResult()`. Metoden samler fortløpende inn resultatene til barna og utfører koden til den opprinnelige rekursive metoden etter det rekursive kallet. Sluttresultatet lagres i arbeiderens egen `result`-variabel.

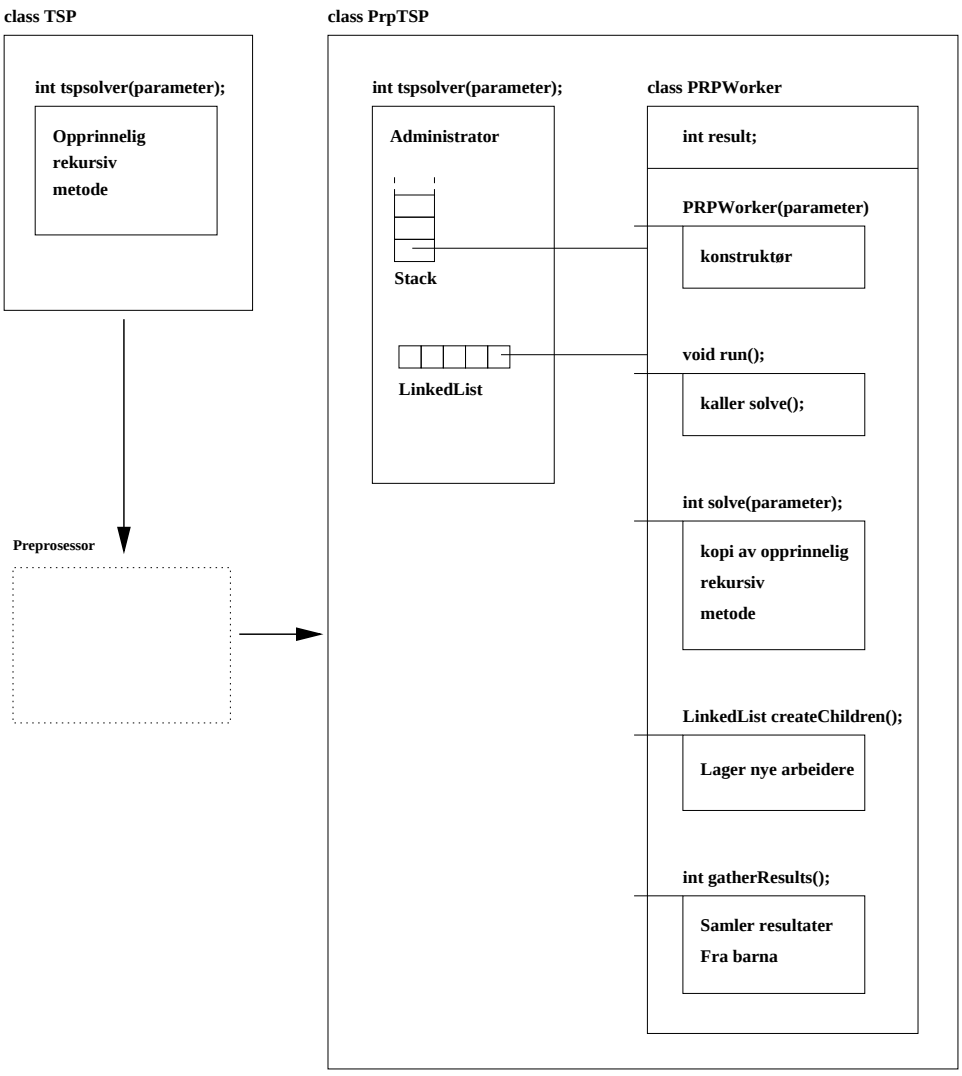
Til slutt beregnes resultatet til den første arbeideren på stakken. Dette er den øverste arbeideren i det rekursive kalltreet, altså den arbeideren som finner svaret på det opprinnelige problemet som skal løses. Resultatet fra denne arbeideren returneres av administratoren, og beregningen er utført.

Denne fremgangsmåten er noe enklere enn den i JavaPRP. Forskjellen er at det denne gangen ikke er nødvendig med noen svararray som tar vare på mellomresultater. Det er fordi alle arbeidere har tatt vare på pekere til sine barn i en liste.

Figur 10.4 på neste side viser et eksempel på preprosessering av et program med rekursiv metode. Man kan se at resultatprogrammet inneholder en rekke nye komponenter i form av en administratormetode og en arbeiderklasse.

10.5 Parallellisering av løkker

Det stilles et viktig krav til hvordan den parallelliserbare løkken skal være implementert. Koden for beregningene som utføres kan ikke plasseres direkte i løkken men må plasseres i en annen metode som løkken kaller på. Selve metoden som inneholder løkken bør inneholde minst mulig kode



Figur 10.4: Preprosessering av programmet Travelling Salesperson. - et eksempel på et program med parallelliserbar rekursiv metode

bortsett fra selve løkken og beregningsfunksjonen. Dette er illustrert i listing 10.4 på neste side. Metode `A` inneholder en løkke hvor det er et kall på metode `B` som utfører en rekke bergninger.

Det som skjer i et program som er bygd opp på denne måten er at metode `B` kalles mange ganger med ulike argumenter. Målet er å eksekvere disse kallene i parallell. MPRP genererer et system som utfører dette ved hjelp av et sett med arbeidere. På samme måte som for rekursive metoder vil arbeiderne representeres av en indre klasse i den genererte koden. En administratorkomponent genereres inne i den opprinnelige metoden som

inneholdt løkken. Et kall på denne metoden blir dermed et kall på administratoren.

Hver arbeider skal ha ansvaret for å utføre en andel av metodekallene. Denne fordelingen av metodekall kan gjøres på flere måter.

En måte kan være å ta for seg hvert enkelt metodekall og lagre argumentene som skal sendes. Det kan gjøres ved å spinne gjennom en variant av løkken der metodekallet er byttet ut med lagring av argumenter. Etterpå kan disse argumentsettene fordeles på en rekke arbeidere som i parallell utfører metodekallene de er blitt tildelt.

Teknikken som er benyttet i MPRP er imidlertid enklere å implementere og krever ingen lagring av argrumentsett.

Listing 10.4: Syntax for parallelliserbar løkke

```
1  /* PRP_LOOP */
   void A() {
     for (int t = 0; t < 1000; t++) {
       /* PRP_CALL */
5      B(t);
     }
   }

   void B(int t) {
10   // gjør beregninger
   }
```

10.5.1 Fordeling av delproblemer

I det genererte programmet har alle arbeiderne en modifisert kopi av den opprinnelige metoden. Den er modifisert slik at kun et utvalg av metodekallene i løkken utføres.

Hvis det er n arbeidere skal altså hver arbeider bare løse $1/n$ av delproblemene (metodekallene). Resten skal hoppes over. Dette utføres ved at samtlige arbeidere i parallell løper gjennom sin egen løkke der de utfører en if-test som vist i listing 10.5 på neste side. `currentProblem` er en teller som angir hvilket delproblem som skal løses, `nWorkers` er lik antall arbeidere og `id` er et unikt nummer som alle arbeiderne er blitt tildelt. Ved at arbeiderne bare løser de delproblemene der

$$id = currentProblem \pmod{nWorkers}$$

sørger man for at de kun utfører hvert n-te kall. Etter at en arbeider har utført et metodekall vil de hoppe over de $n - 1$ neste. Arbeiderne løser aldri identiske metodekall, og på denne måten vil arbeiderne til sammen løse alle delproblemene.

Det kan virke som om det er ressurskrevende at samtlige arbeidere løper gjennom hele løkken på denne måten. Men så lenge beregningstyngden

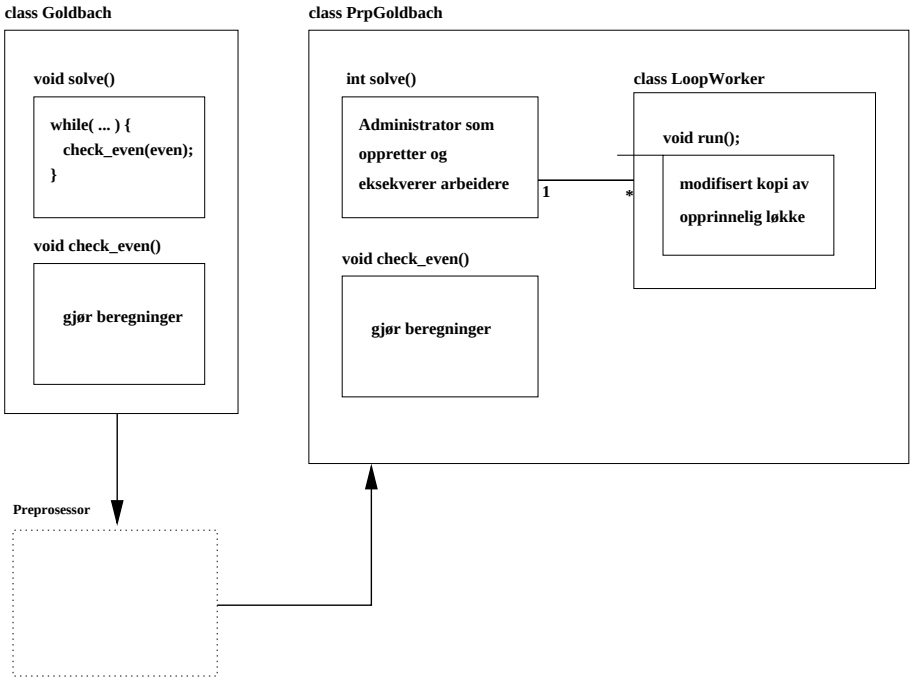
i disse programmene ligger i metoden som kalles fra løkken er ikke dette noe problem. Den tiden en arbeider bruker på å hoppe over $n-1$ metodekall er uvesentlig i forhold til den tiden den bruker på å løse det påfølgende delproblem.

Listing 10.5: Utvelgelse av delproblemer

```
1  currentProblem=0;
   for( ... ) {
     currentProblem++;
     if (id == currentProblem % nWorkers) {
5      solveProblem();
     }
   }
```

10.5.2 Eksekvering

Figur 10.5 viser resultatet etter preprosessering av programmet Goldbach. Programmet inneholder en metode `solve` som inneholder en løkke som brukeren ønsker å parallellisere. Inne i løkken er det et kall på metoden `check_even` som utfører en rekke beregninger.



Figur 10.5: Preprosessering av programmet Goldbach - et program med parallelliserbar løkke

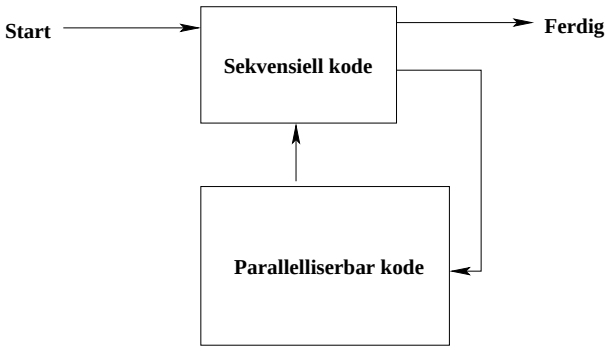
Som vist på figuren blir metoden `solve` byttet ut med en administratormetode. Metoden sjekker hvor mange prosessorer som er tilgjengelig på maskinen og oppretter et antall arbeidere i forhold til det. Arbeiderne representeres som instanser av den indre klassen `LoopWorker`.

`LoopWorker` implementerer interfacet `Runnable` slik at instansene av denne kan tilordnes hver sin tråd og eksekveres i parallell. Ved eksekvering kalles dermed metoden `run()`.

Metoden `run()` inneholder den modifiserte utgaven av den opprinnelige metoden. Den inneholder altså en løkke som gjør et kall på en annen metode. Som beskrevet over er imidlertid metoden modifisert slik at hver arbeider kun utfører et utvalg av kallene.

10.6 Gjentatt parallelliserbar løkke

For noen problemer er det nødvendig med flere parallelle deler som må eksekveres uavhengig og sekvensielt etter hverandre. Jeg kaller disse algoritmene pumpealgoritmer. Et eksempel på et problem som løses ved hjelp av en slik pumpealgoritme er Optimalt Søketre. Det omtales i kapittel 8 på side 45.



Figur 10.6: Sekvensdiagram for pumpealgoritmer

Et program som benytter seg av pumpealgoritme er strukturert som på figur 10.6. Programmet består av en parallelliserbar del og en sekvensiell del. Den parallelliserbare delen kan bestå av en løkke eller en rekursiv metode. Under kjøring hopper programmet inn og ut av den parallelliserbare delen. Dette skjer ved hjelp av en løkke i den sekvensielle delen. Programmet går altså i en sekvensiell løkke, og ved hver iterasjon utfører det parallelliserbar kode. Man kan se på den sekvensielle delen som en pumpe som gjentatte ganger pumper problemer ut til den parallelle delen.

10.6.1 Nytt system for hvert parallelt problem

Det finnes flere måter å håndtere disse algoritmene på. Et alternativ er å ignorere den sekvensielle delen og håndtere hvert enkelt kall på den parallelle delen individuelt. Det vil si at hver gang programmet går inn i den parallelle delen initialiseres et helt nytt parallelt system med administrator og arbeidere. Systemet utfører alle beregninger og returnerer tilbake til sekvensiell del. Det sekvensielle programmet vil så fortsette å iterere i sin løkke og gjøre nye kall på parallell del slik at det hele gjentas. Dette betyr at det parallelle systemet må opprettes mange ganger under eksekveringen. Det tar tid, og fordi det må skje mange ganger kan det være en ulempe. Fordelen med dette alternativet er at det er enkelt å implementere.

10.6.2 Gjenbruk av parallelt system

Et annet alternativ er å opprette det parallelle systemet med administrator og arbeidere på forhånd. På den måten kan programmet bruke dette samme systemet hver gang det går inn i den parallelle delen. Det vil altså opprettes et enkelt sett med arbeidere som administratoren bruker flere ganger. Administratoren har et sett med parallelle problemer som må løses i sekvensiell rekkefølge. Den tar for seg et og et problem som så løses i parallell av arbeiderne.

Denne håndteringen av delproblemer kan foregå på flere ulike måter. Kapittel 8 på side 45 tar for seg ulike pumpealgoritmer for optimalt søketre. Det viste seg at barriersynkronisering var mest lønnsomt. Det tyder på at det er hensiktsmessig å benytte barriersynkronisering også i MPRP.

Listing 10.6: Barriersynkronisering blant arbeiderne

```
1  class LoopWorker {
     run {
       //sekvensiell løkke
       while( ... ) {
5        //Løs et delproblem
         A();
         //vent på alle andre arbeidere
         barrier.wait();
       }
10   }

     /* PRP_LOOP */
     void A() {

15   }
   }
```

En implementasjon av arbeidere med barriersynkronisering vil kunne se ut som i listing 10.6. Alle arbeiderne får en kopi av den sekvensielle pumpeløkka. På slutten av løkka er det lagt inn et kall på `barrier.wait()`

slik at arbeiderne utfører en barriersynkronisering. Inne i løkka finnes et kall på arbeiderens egen skreddersydde kopi av PRP-løkken, som utfører arbeiderens egne tilordnede problemer. Etter oppstart går alle arbeidere inn i sin sekvensielle løkke. Inne i løkken utfører de kall på sin PRP-løkke for å løse et delproblem. Men etter hver iterasjon møter de en barrier, slik at de må vente til alle de andre arbeiderne er ferdig med sine delproblemer før de kan fortsette.

10.6.3 To alternative fremgangsmåter

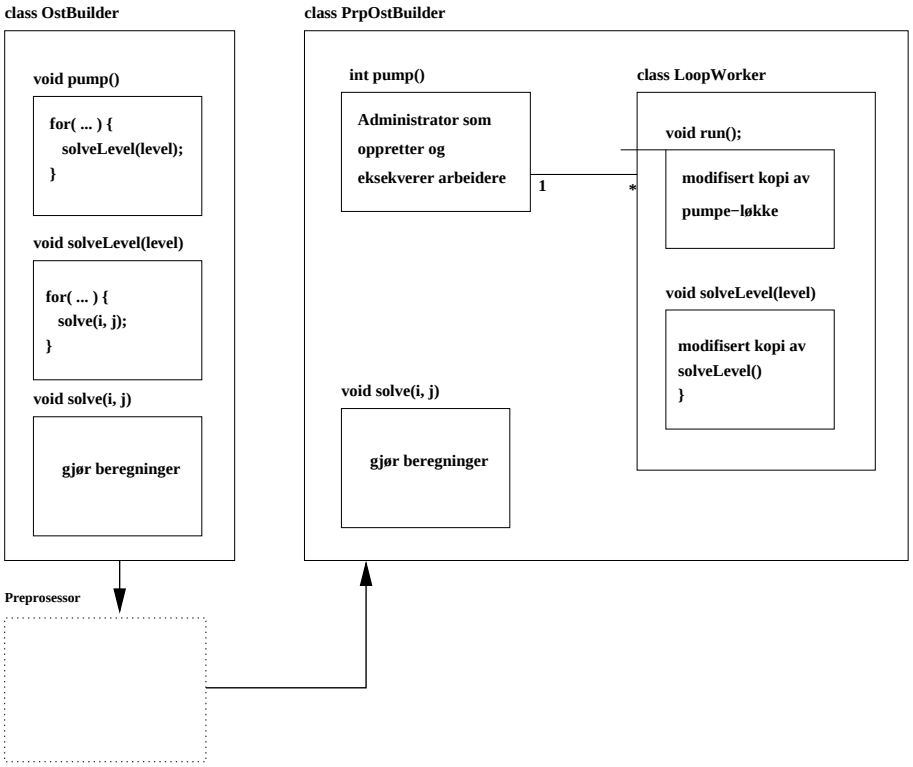
Vi står igjen med to mulige fremgangsmåter for å løse pumpeproblemer. Det første forslaget er å håndtere hvert enkelt parallelt problem separat. Dette er den fremgangsmåten vi hadde stått igjen med hvis vi aldri hadde tatt hensyn til pumpealgoritmene. Det medfører at en ny administratormetode kalles på nytt for hver gang pumpeløkken vil løse et problem. Denne fremgangsmåten krever altså lite eller ingen ekstra implementasjon.

Den andre fremgangsmåten dreier seg om gjenbruk av det parallelle systemet. Denne er mer komplisert både implementasjonsmessig og i bruk. Det er mer komplisert i bruk fordi brukeren blir nødt til å fortelle preprosessoren hva som er pumpemetoden. Den kan typisk markeres med en kommentar på formen `/* PUMP_LOOP */`. Implementasjonsmessig innebærer dette at disse algoritmene må håndteres på en spesiell måte. Strukturen i resultatprogrammet blir annerledes.

Fordi spesialhåndteringen ved hjelp av barriersynkronisering viste seg å gi en god ytelsesforbedring i implementasjonen av optimalt søketre har det blitt valgt å implementere støtte for dette i MPRP. Dette krever at brukeren har gitt beskjed om at det dreier seg om en pumpealgoritme ved å markere pumpemetoden. Dersom pumpe-metoden ikke markeres vil pumpen ignoreres slik at den første fremgangsmåten egentlig benyttes. På den måten støttes begge fremgangsmåter. Det er opp til brukeren å bestemme hvilken som skal benyttes.

10.7 Resultatsystem

Resultatet av dette arbeidet er et program som ved hjelp av et grafisk brukergrensesnitt tilbyr preprosessering av Java-programmer. I tillegg A på side 85 blir det forklart hvordan systemet brukes. Tillegg B på side 101 viser eksempler på programmer som er preprosessert av systemet. Det vises kildekode fra programmer der det er benyttet rekursjon, løkke og gjentatt parallelliserbar løkke.



Figur 10.7: Preprosessering av programmet OstBuilder - et program med pumpealgoritme

Kapittel 11

Oppsummering og videre arbeid

11.1 Konklusjon

Målet med denne oppgaven har vært å finne ut hvordan man på en god måte kan utføre automatisk parallellisering på multiprosessor. Både fordelene og utfordringene ved multiprosessoren skulle håndteres på en best mulig måte.

Som forberedelse til automatisk parallellisering ble det først sett på hvordan man kan utføre manuell parallellisering. En utfordring var at de mange algoritmene som man kan tenke seg å parallellisere virker på ulike måter. For eksempel er det forskjell på algoritmer som benytter rekursjon og algoritmer som benytter løkker. For å finne ut hvordan ulike typer algoritmer best mulig bør parallelliseres har det blitt sett på tre ulike problemer, og parallelliseringen av disse.

Algoritmen til det første problemet benytter rekursjon, og parallelliseringen har blitt inspirert av teknikker i JavaPRP. I algoritmen til det andre problemet var det en løkke som skulle parallelliseres, noe som kunne håndteres på en enklere måte enn rekursjon. Det tredje problemet hadde overlappende delproblemer, og det ble funnet ut at man ved hjelp av barriersynkronisering kunne effektivisere parallelliseringen av dette.

Sammenlignet med systemer som JavaPRP der parallelliseringen foregår over nettverk av maskiner er det relativt enkelt å parallellisere på multiprosessor. Ved å la et program opprette mange tråder kan det ved hjelp av disse utføre beregninger i parallell.

Den største fordelen ved parallellisering på multiprosessor er den delte hukommelsen. Trådene som benyttes i parallelle programmer for multiprosessor kan på en enkel måte få tilgang til delt hukommelse ved å aksessere felles variabler. Ved bruk av dette er det imidlertid ulike problemer som

kan oppstå, og disse bør håndteres ved hjelp av synkroniseringsmekanismer. De ulike parallelle programmene som har blitt presentert benytter felles hukommelse og håndterer problemene knyttet til dette på ulike måter.

Videre har det blitt drøftet hvordan man kan utføre automatisk parallellisering på multiprosessor. En mulighet var å benytte JavaPRP, som har støtte for utnyttelse av multiprosessor ved at det blir startet flere uavhengige arbeidere på multiprosessoren. Dette viste seg å være en lite hensiktsmessig løsning fordi JavaPRP kun har støtte for rekursjon, og ingen støtte for delt hukommelse. Dessuten er JavaPRP unødvendig tungvindt i bruk når man kun har å gjøre med én maskin.

En annen mulighet var å utvide JavaPRP til å bedre håndtere alt dette. Det ville imidlertid ha vært en stor jobb, og det ville trolig ha vært vanskelig å gjøre noe med brukervennlighetsaspektet.

Løsningen som derfor ble valgt var å utvikle et helt nytt system som skulle ta seg av generering av parallelle programmer for multiprosessor. Systemet MPRP er i stand til å utføre preprosessering av programmer der en av de tre programmeringsteknikkene nevnt over er benyttet. MPRP er derfor mer fleksibelt enn JavaPRP fordi det har støtte for flere programmeringsteknikker. Fordi de tre ulike teknikkene bør håndteres på ulike måter under parallelliseringen var det mest hensiktsmessig å implementere tre ulike mekanismer for disse. Systemet velger mekanisme avhengig av hvilken teknikk som er valgt basert på de kommentarer som er lagt inn i koden.

Det har blitt valgt å la brukeren fritt bestemme hvordan de genererte parallelle programmene skal benytte seg av delt hukommelse. Brukeren implementerer dette ved å benytte globale variabler i det sekvensielle programmet. Disse mekanismene vil dermed automatisk overføres til det parallelle. Dette medfører at brukeren selv må sørge for sikkerthet rundt de problemene som kan oppstå knyttet til dette, men det gir også en høy grad av fleksibilitet.

Kort oppsummert tilbyr preprosessoren MPRP en enkel og fleksibel behandling av sekvensielle programmer slik at multiprosessorens egenskaper utnyttes godt. I tillegg B på side 101 kan man se at de genererte parallelle programmene er mye større enn de opprinnelige. Det viser at det trolig er klart enklere å lage parallelle programmer ved hjelp av MPRP enn å manuelt programmere slike.

11.2 Svakheter ved oppgaven

I arbeidet med oppgaven har jeg sett at det er visse ting som muligens kunne vært gjort annerledes. Det er særlig tre aspekter jeg vil ta for meg her.

11.2.1 Navn på systemet

Navnet MPRP er avledet av forkortelsen PRP (Parallel Recursive Procedures) og ble valgt fordi denne oppgaven er en del av PRP-prosjektet. Navnet er imidlertid noe missvisende, da systemet ikke bare er i stand til å parallellisere rekursjon men også andre programmeringsteknikker. Andre forkortelser som MPM eller MPP kunne kanskje være bedre egnet.

11.2.2 Raskere algoritmer

For noen av problemene kunne jeg valgt raskere sekvensielle algoritmer som utgangspunkt for parallelliseringen. Både for Goldbach og Travelling Salesperson finnes det algoritmer som er mye raskere enn de som er brukt i denne oppgaven. Dette er imidlertid lite relevant for selve parallelliseringen. Målet her var å demonstrere parallellisering, og da kan det være hensiktsmessig å benytte enkle algoritmer som eksempler.

11.2.3 Fokus på felles hukommelse

Jeg kunne også hatt større fokus på håndtering av felles hukommelse. I denne oppgaven blir det forklart hvordan felles hukommelse fungerer og bør håndteres av parallelle programmer, men i MPRP har det ikke blitt implementert noen mekanismer som skal ta seg av akkurat dette. Det er overlatt til brukeren å sørge for at felles hukommelse håndteres på en fornuftig måte i de genererte parallelle programmene.

Kanskje ville det vært fornuftig å bare fokusere på én type problemer, for eksempel bare rekursjon, og heller fokusere på god utnyttelse av felles hukommelse for akkurat den typen problemer.

11.3 Forslag til videre arbeid

Gjennom arbeidet med oppgaven har det vært mange nye problemstillinger som har dukket opp. Flere av disse har det ikke vært tid til å fokusere på i denne oppgaven, og dette avsnittet gir derfor noen forslag til hva andre kan fokusere på i sine påfølgende arbeid.

11.3.1 Bedre håndtering av felles hukommelse

I dagens versjon av MPRP må brukeren selv ta seg av all håndtering av felles hukommelse. Brukeren må opprette globale variabler og sørge før at skriving til og lesing fra disse foregår på en sikker måte. Kanskje er det mulig å implementere noe som tar seg av dette automatisk. I en slik løsning kan det benyttes nye kodeord for å markere delte variabler. Preprosessoren

kan da opprette mekanismer i programmet som sørger for at de delte variablene blir brukt på en sikker måte. En annen mulig utvidelse er at MPRP under preprosesseringen kontrollerer om programmets håndtering av delt hukommelse vil komme til å fungere, og gir advarsler dersom det er fare for at problemer kan oppstå.

11.3.2 Integrasjon i JavaPRP

På sikt kan det være aktuelt å utvide JavaPRP til å ha bedre støtte for multiprosessor. En mulighet vil da være at arbeidere som kjører på samme prosessor kan benytte delt hukommelse. Dette vil gi en slags hybridløsning fordi arbeidere på ulike maskiner fortsatt må kommunisere over nettverket. Kanskje har arbeidet i denne oppgaven gitt inspirasjon til hvordan en slik løsning kan implementeres.

11.3.3 Testing på flere typer maskiner

I denne oppgaven har de parallelle programmene kun blitt testet på en maskin med 64-bits Linux og firekjerneprosessor fra Intel. Det ville være interessant å finne ut om andre typer prosessorer eller operativsystemer gir ulike resultater. Andre operativsystemer kan være ulike versjoner av Windows eller Mac OS X. Andre maskiner kan være prosessorer fra andre selskaper som AMD eller fremtidige prosessorer med flere kjerner. Det ville også ha vært interessant å forsøke parallellisering på prosessorer med spesiell og nyskapende arkitektur, som for eksempel Cell-prosessoren fra IBM.

11.3.4 Brukergrensesnitt

MPRP har et enkelt brukergrensesnitt med få funksjoner. Det vil være mulig å utvide dette både til å tilby flere funksjoner og til å gi mer informasjon. For eksempel kan det være ønskelig å kunne starte resultatprogrammet direkte fra det grafiske brukergrensesnittet. Under kjøring av resultatprogrammet kunne det ha vært mulig å få mer informasjon om programmets håndtering av ressursene på datamaskinen.

Feilhåndteringen til preprosessoren er også begrenset. Det kunne være ønskelig at det ble gitt tydeligere og bedre forklaringer dersom det skulle oppstå feilsituasjoner under preprosesseringen.

11.3.5 Færre krav til inputprogrammet

MPRP stiller færre syntaktiske krav til inputprogrammet enn det preprosessoren i JavaPRP gjør. Likevel er det fortsatt mange krav som trolig kan fjernes. For eksempel kan det trolig være mulig å tillate flere rekursjonskall

i en rekursiv metode. Aller helst bør det nesten ikke være noen krav, slik at så godt som hvilket som helst program kan preprosesseres.

Tillegg A

User guide to MPRP

This guide will explain how MRP (Multicore Parallel Recursive Procedures) works, and it will show an example of usage.

Figure A.1: Model of how MPRP works

MPRP is a tool used for generating parallel programs. It will automatically generate a parallel program based on a sequential program that is annotated with a few comments by the user.

The generated program has the same functionality and signature as the original one, the difference is that the code will be modified so that the instructions will execute in parallel.

Inside the sequential program there must be a parallelizable section (annotated by the user). The generated program uses multiple threads to execute that section, and should be able to run faster than the sequential program if run on a multiprocessor.

A.1 Three programming techniques

The input program can be an almost ordinary java program, although some of it has to be parallelizable. MPRP is capable of parallelizing three different programming techniques. These are **recursion**, **loop** and **repeated loop**. The program has to be annotated differently according to which technique is used in the program.

A.1.1 Recursion

If the code to be parallelized is a recursive method, the method itself and the recursive call has to be annotated. The following two keywords are used:

- `/* PRP_PROC */`
  marks the recursive method. The comment is inserted on the line directly above the method's header.
- `/* PRP_CALL */`
  marks the recursive call. The comment is inserted on the line directly above the expression containing the call.

For the parallelization to work, the method must in average call itself minimum twice. To achieve this, the call must be inside a loop. Listing A.1 shows an example of a recursive method like this.

Listing A.1: Recursive method

```
1  /* PRP_PROC */
   int solveProblem( ... ) {
     for( ...) {
       //do calculations recursively
5      /* PRP_CALL */
       solveProblem( ... );
     }
     return answer;
   }
```

Besides from adding the annotations it is not expected that the user needs to change the original program when parallelizing recursion.

A.1.2 Loop

MPRP can also parallelize a loop. To do so the programmer needs to modify the program in a trivial way. The block in the loop must be placed alone inside a separate method, and inside the original loop there must be a call this separate method where the actual calculations are done. Two keywords must be used to annotate the loop:

- `/* PRP_LOOP */`
  marks the original method containing the loop. The comment is inserted on the line directly above the method's header.
- `/* PRP_CALL */`
  marks a method call inside the original loop. The comment is inserted on the line directly above the expression containing the call.

Listing A.2 on the facing page shows an example of this kind of loop.

Listing A.2: Method with loop

```
1  /* PRP_LOOP */
   void loopMethod( ... ) {
     for ( ... ) {
       /* PRP_CALL */
5      solveProblem(i);
     }
   }

   void solveProblem(i) {
10   // do calculations
     // the content of the loop must be placed here
   }
```

It is important that all the actual calculations are being done inside the method called by the loop. In listing A.2 the calculations will be done inside `solveProblem()`. Besides from the call to the method (`solveProblem`) the loop (in `loopMethod`) should contain as few statements as possible as these would be executed in parallel by all threads in the generated program. Be aware that the names of these methods are arbitrary, and can be chosen freely by the programmer.

A.1.3 Repeated parallelizable loop

Some programs are constructed so that the parallelizable method in the program is called several times by another method. In this case the other method usually has a loop which causes the parallelization to start several times during execution. Figure A.2 on the following page illustrates this. We call the method containing the sequential loop a **pump**. It is possible to generate a faster parallel program if the programmer modifies the sequential program in a trivial way and uses additional keywords to annotate the pump. This will make the parallel program execute faster because it will be able to reuse the parallel system every time the parallelizable method is called.

The parallelizable loop must be modified and annotated in the same way as a regular parallelizable loop as described in the previous section. In a similar way the pump loop must be placed in a separate method, where both the method and the pump call is annotated. The following keywords are used:

- `/* PUMP_LOOP */`
  marks the pump method. The comment is inserted on the line directly above the method's header.

- `/* PUMP_CALL */`
  marks the call to the parallelizable method inside the pump. The

comment is inserted on the line directly above the expression containing the call.

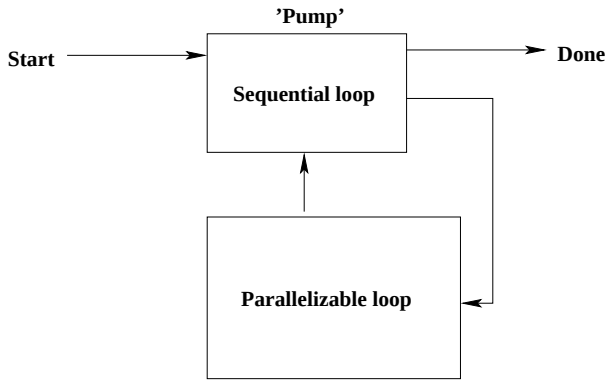


Figure A.2: Sequence diagram for repeated parallelizable loop

Listing A.3 shows an example of a pump loop. The method `sequentialLoop()` contains a loop which must be executed sequentially. Inside the sequential loop there is a call to the method `loopMethod()` which has a parallelizable loop inside. `loopMethod()` and `solveProblem()` are identical to the methods in listing A.2 on the preceding page. The only difference here is that another method (`sequentialLoop()` is added which repeatedly makes calls to the parallelizable loop. The names of the methods are arbitrary.

Listing A.3: Loop parallelization with pump

```
1   /* PUMP_LOOP */
    void sequentialLoop( ... ) {
      for( ... ) {
        /* PUMP_CALL */
5       loopMethod();
      }
    }

    /* PRP_LOOP */
10  void loopMethod( ... ) {
      for ( ... ) {
        /* PRP_CALL */
        solveProblem(i);
      }
15  }

    void solveProblem(i) {
      // do calculations
    }
```

A.2 Requirements to the input program

The input program must fulfill the following requirements:

- **One class in one file**
  MPRP will do preprocessing on only one file. The file can only contain one class.
- **Runnable input program** The provided program has to be compilable and runnable as a sequential program.
- **Parallelizable method**
  The parallelizable part of the program must be placed inside an object method in the class mentioned above, so the method cannot be static. The method and its content must be annotated with the keywords described in section A.1 on page 85. There can only be one parallelizable method like this in the program.
- **Name restrictions**
  When the program is generated the preprocessor will add certain variables, methods and inner classes. These will have names that are prefixed with `Prp` (For example `PrpWorker`). To avoid conflicting names it is important that the provided program does not contain names beginning with this prefix.

A.3 Shared memory

When making parallel programs with MPRP the user may take use of the shared memory available on a multiprocessor. This can be achieved by letting the parallelizable method do calls to other object methods or object variables. In the generated program these methods and variables can be considered as global among the threads, so calls will be executed by the threads in parallel. When using shared memory like this it is important to handle the problems related to asynchronous cache and race conditions. The problems should be handled differently depending how the shared memory is being used. The following scenarios may occur:

1. **Data will not change during parallel fase**
   If the parallelizable code only does *reads* to global variables, no problems will occur. There are no need to use synchronization mechanisms.
2. **Data is updated during parallel fase**
   If there are executed *writes* to the global variables, synchronization mechanisms may have to be used during both writing and reading:

(a) **Writing**
In most cases synchronization must be used by writing through a synchronized method. Listing A.4 shows an example of this. An exception is if there are no reads and it is guaranteed that the threads never write to the exact same variables.

Listing A.4: Writing to a shared variable

```
1  int val;

   synchronized void setVal(int newval) {
     val = newval;
5  }
   /* PRP_PROC */
   void proc() {
     ...

10    setVal(4);

   }
```

(b) **Reading**
To guarantee that the last updated value is retrieved when reading a variable, synchronization primitives must be used. In most cases this can be done by using a synchronized method. For simple variables it can also be achieved by declaring them as `volatile`. Listing A.5 shows examples of both mechanisms.

Listing A.5: Reading from shared variables

```
1  int val;
   volatile int val2;

   synchronized int getVal() {
5    return val;
   }

   /* PRP_PROC */
   void proc() {
10   ...

     x = getVal();
     x2 = val2;
   }
```

Another option is to use classes in the package `java.util.concurrent.atomic` which offers atomic and volatile operations on variables or arrays.

The sample program in the next section shows an example of shared memory usage, and demonstrates the parallelizable method calling a synchronized method used for writing to a shared variable.

A.4 Example usage

This section presents a step by step guide showing how to preprocess an example program. The program used is a simple program called `SumMatrix`. The purpose of this program is to calculate the sum of all the integers in a 2D matrix. This is done by looping over all the rows in the matrix. For each row a method (`sumRow()`) is called which calculates the sum of all the integers in that row. That loop is parallelizable. The method `sumRow()` will add it's result to a global variable, which will eventually contain the sum of the entire matrix.

Preparation of input file

In the file **SumMatrix.java** we find the parallelizable loop shown in listing A.6.

Listing A.6: The loop to parallelize

```
1
  void sum(int[][] m) {
    for (int[] row : m) {

5     sumRow(row);
    }
  }
```

The loop is placed alone inside a method. Inside the loop there is a method call which is executed several times, and this is what we want to parallelize.

For MPRP to find the loop it has to be annotated with comments. The comment `/* PRP_PROC */` is being used to mark the method and the comment `/* PRP_CALL */` is being used to mark the method call inside the loop.

Listing A.7 on the next page shows the entire program after the comments are added. The program is now complete and ready for preprocessing.

Notice the object variable `sum` which is used to summarize the final result. The method `sumRow()` is increasing the value of this variable by using the synchronized method `increaseSum()`. The method is synchronized because it will be called in parallel by multiple threads in the generated program.

Add annotations in the input program as described earlier.

Listing A.7: SumMatrix.java

```
1  package prpprojects;

   public class SumMatrix {

5    private int sum = 0;

     synchronized void increaseSum(int d) {
       sum += d;
     }
10
     /* PRP_LOOP */
     void sum(int[][] m) {
       for (int[] row : m) {
         /* PRP_CALL */
15       sumRow(row);
       }
     }

     //sums one row and adds the result to global variable sum
20   void sumRow(int[] row) {
       int rowSum = 0;
       for (int num : row) {
         rowSum += num;
       }
25     increaseSum(rowSum);
     }

     public static void main(String[] args) {
       int[][] m = { { 5, 3, 2, 2, 5, 7 },
30       { 2, 3, 2, 7, 3, 3 },
         { 6, 5, 4, 1, 7, 3 },
         { 4, 2, 6, 4, 2, 7 },
         { 4, 6, 7, 3, 2, 6 },
         { 3, 5, 7, 3, 1, 6 },
35       { 4, 6, 7, 3, 2, 5 },
         { 4, 2, 6, 8, 3, 1 } };

       SumMatrix sa = new SumMatrix();
       sa.sum(m);
40     System.out.println("sum: " + sa.sum);
     }
   }
```

Save the file (SumMatrix.java).

Startup

MPRP is distributed in form of the executable file **MPRP.jar**. This is shown in figure A.3 on the facing page together with the program we want to preprocess. The files may be located anywhere on the system.

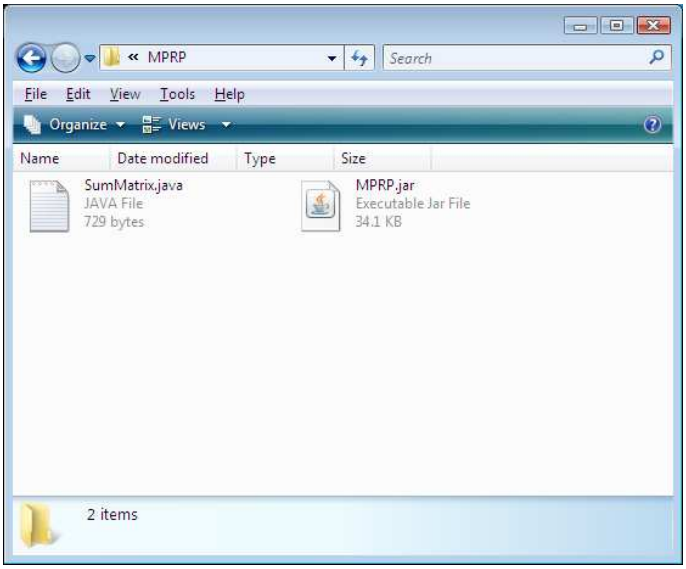


Figure A.3: Catalog with the files

Double-click on MPRP.jar to start the program.

Functionality

When the program starts, the main window will appear. See figure A.4.

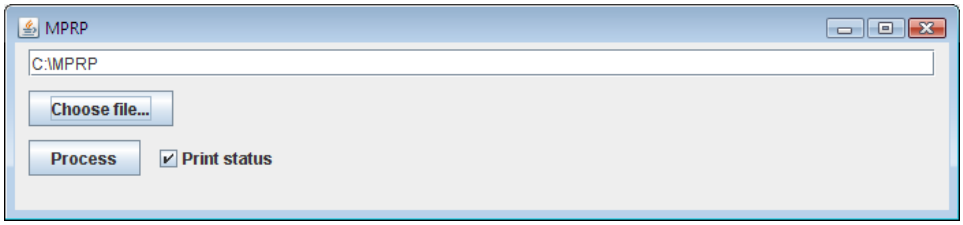


Figure A.4: Main window

The window contains the following:

- A address field for entering the location of the file we want to process.
- A button **'Choose file...'** which opens an explorer used to navigate to the correct file.
- A button **'Process'** for starting the preprocessing.

- A checkbox **'Print status'**. Mark this box to tell MPRP to add code for status output in the program. The resulting program will then output information about the parallelization during execution.

Choosing input file

Click on **'Choose file...'**.

An explorer window (figure A.5) will appear.

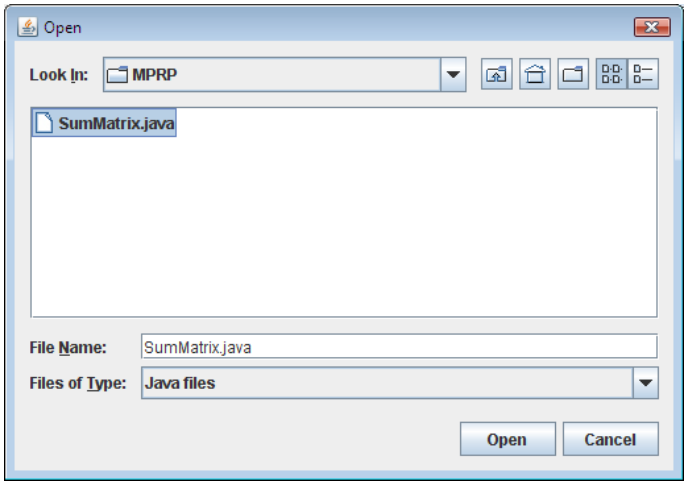


Figure A.5: Choosing input file

Choose the file (in this case SumMatrix.java) and click on **'Open'**.

Processing

The address in the address field should now show a path pointing to the location of the input file.

Click on **'Process'** to start the preprocessing.

The program will now execute two tasks:

1. Generation of the parallel program. If this generation is successful the text **'Generating code...done!'** will appear.

2. Compilation of the parallel program. If this compilation is successful the text **'Compiling...done!'** will appear.

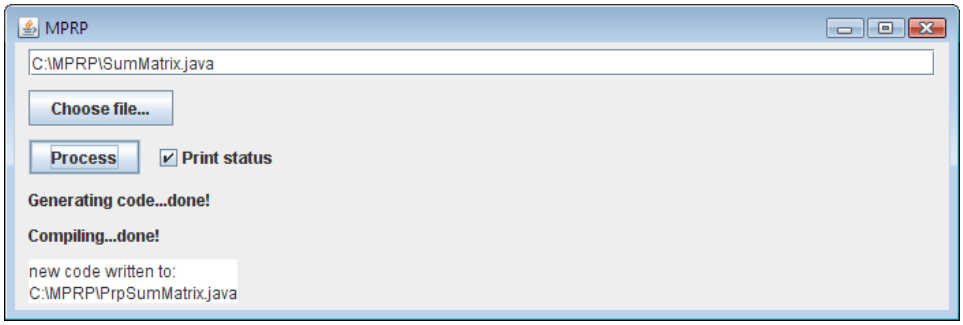


Figure A.6: Main window after preprocessing

If everything went OK the main window will look like the one shown in figure A.6.

At the bottom of the window the path to the location of the generated program is shown.

Generated files

During the processing a file was generated containing source code for the parallel program as well as compiled files. The new files are stored in the same catalog as the input program as shown in figure A.7 on the following page. The new source code file has been given the same name as the original program with the prefix **Prp** (in this case **'PrpSumMatrix.java'**).

Execution

The parallel program is now ready to be run. Listing A.8 on page 97 shows the generated source code. The generated parallel program has the same functionality and the same signature as the original program with the exception of the parallel execution of the calculations.

The parallel program can be executed in the same way as the original. Figure A.8 on the following page shows an execution of PrpSumMatrix.java on a multiprocessor capable of executing eight parallel threads.

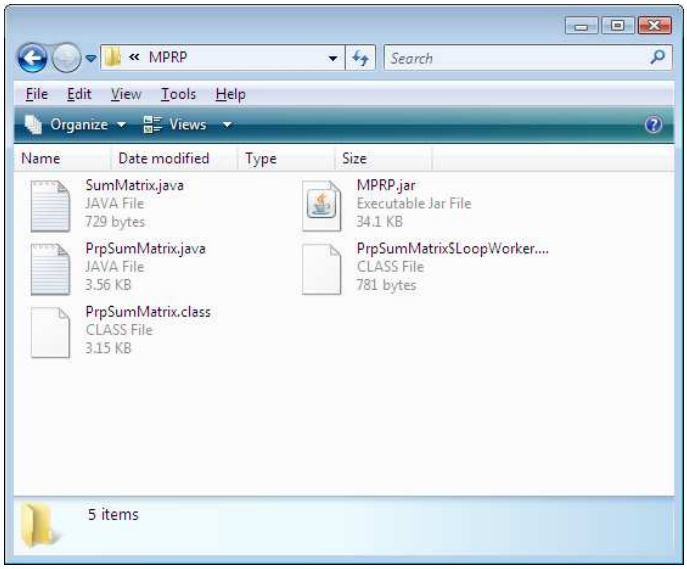


Figure A.7: Generated files

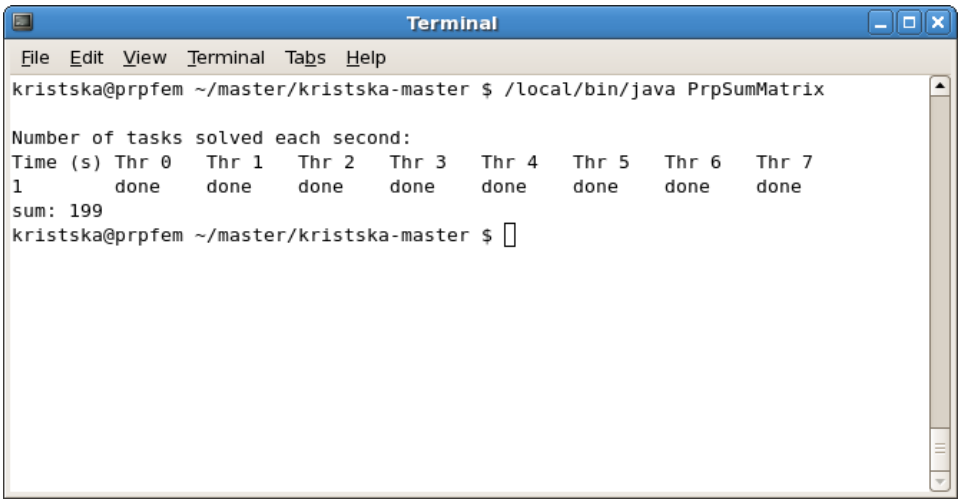


Figure A.8: Execution of the result program under linux.

Generated code

Listing A.8: PrpSumMatrix.java

```
1  import java.util.Iterator;
   public class PrpSumMatrix {

     private int sum = 0;
5
     synchronized void increaseSum(int d) {
       sum += d;
     }

10   /* PRP_LOOP */
     void sum(int[][] m) {
       int nCPUs = Runtime.getRuntime().availableProcessors();
       int nThreads = nCPUs;
       barrier = new CyclicBarrier(nThreads);
15
       // store the LoopWorkers in an ArrayList instead of an array in case of
       // generic class. Array of generic class does not work
       ArrayList<LoopWorker> workers = new ArrayList<LoopWorker>(nThreads);
       ExecutorService executor = Executors.newFixedThreadPool(nCPUs);
20
       for (int cpu = 0; cpu < nThreads; cpu++) {
         LoopWorker l = new LoopWorker(m, cpu, nThreads);
         executor.execute(l);
         workers.add(l);
25     }

       // print status header
       final int colWidth = 8;
       System.out.println("\nNumber␣of␣tasks␣solved␣each␣second:");
30     System.out.print("Time␣(s)␣");
       for (LoopWorker lw : workers) {
         String header = "Thr␣" + lw.id + "␣␣␣␣";
         System.out.print(header.substring(0, colWidth));
       }
35     System.out.println();

       try {
         executor.shutdown();
         int time = 0;
40       while (!executor.isTerminated()) {
           // wait 1 second
           synchronized (this) {
             executor.awaitTermination(1, TimeUnit.SECONDS);
           }
45         time++;
           String timeString = String.valueOf(time);
           System.out.print(time);
           for (int space = timeString.length(); space < 9; space++)
           System.out.print("␣");
50         // print some status info
```

```
        for (LoopWorker lw : workers) {
          if (lw.status > 0) {
            int tasks = lw.status - lw.oldstatus;
            lw.oldstatus = lw.status;
            String taskstring = String.valueOf(tasks);
            System.out.print(taskstring);
            for (int space = taskstring.length(); space < colWidth; space++)
            System.out.print(" ");
          } else if (lw.status == 0) {
            System.out.print("waiting  ");
          } else if (lw.status < 0) {
            System.out.print("done    ");
          }

        }
        System.out.println();
      }
    } catch (InterruptedException e1) {
      e1.printStackTrace();
    }
  }

  CyclicBarrier barrier;

  private class LoopWorker implements Runnable {
    int[][] m;

    int id;

    int nWorkers;

    int currentWorker;

    int status;

    int oldstatus;

    public LoopWorker(int[][] m, int id, int nWorkers) {
      this.m = m;
      this.id = id;
      this.nWorkers = nWorkers;
      this.currentWorker = 0;
      this.status = 0;
      this.oldstatus = 0;
    }

    public void run() {

      for (int[] row : m) {
        /* PRP_CALL */
        currentWorker++;
        if (currentWorker % nWorkers == id) {
          sumRow(row);
          status++;
```

```
105             }

            }

            status = -1;
110       }
      }

      // sums one row and adds the result to global variable sum
      void sumRow(int[] row) {
115     int rowSum = 0;
        for (int num : row) {
          rowSum += num;
        }
        increaseSum(rowSum);
120   }

      public static void main(String[] args) {
        int[][] m = { { 5, 3, 2, 2, 5, 7 },
          { 2, 3, 2, 7, 3, 3 },
125       { 6, 5, 4, 1, 7, 3 },
          { 4, 2, 6, 4, 2, 7 },
          { 4, 6, 7, 3, 2, 6 },
          { 3, 5, 7, 3, 1, 6 },
          { 4, 6, 7, 3, 2, 5 },
130       { 4, 2, 6, 8, 3, 1 } };

        PrpSumMatrix sa = new PrpSumMatrix();
        sa.sum(m);
        System.out.println("sum: " + sa.sum);
135   }
    }
```

Tillegg B

Kildekode

Dette kapittelet viser eksempler på preprosessert kildekode. Kapittelet viser kildekoden til tre ulike sekvensielle programmer samt de genererte parallelle utgavene av programmene.

B.1 Traveling Salesperson

Listing B.1: TSP.java

```
1  package prpprojects;

   import java.util.Random;

5  public class TSP {

     public int[][] problem; // problem[i][j] is distance from i to j and

     // from j to i
10
     public volatile int cutoff;

     public TSP(int[][] problem) {
       this.problem = problem;
15     this.cutoff = Integer.MAX_VALUE;

     }

     /**
20   * creates a random problem, and initializes TSP
     *
     * @param args
     */
     public static void main(String[] args) {
25     long start = System.currentTimeMillis();
       if (args.length < 1) {
```

```
      System.out
      .println("Usage: java TSP <size> (<random seed>) (<edge density>)");
      System.exit(0);
    }
    int size = Integer.parseInt(args[0]);

    int seed = 1;
    if (args.length >= 2) {
      seed = Integer.parseInt(args[1]);
    }
    double density = 1;
    if (args.length >= 3) {
      density = Double.parseDouble(args[2]);
    }

    int[][] problem = randomProblem(size, seed, density);
    problem = fixTwoWayEdges(problem);

    // print problem
    System.out.println("problem:");
    for (int i = 0; i < problem.length; i++) {
      for (int j = 0; j < problem.length; j++) {
        System.out.print(problem[i][j] + " ");
      }
      System.out.print("\n");
    }
    TSP solver;

    solver = new TSP(problem);

    solver.solve();

    long end = System.currentTimeMillis();
    long time = end - start;
    System.out.println("Total time " + time + " ms ");
  }

  private static int[][] randomProblem(int size, long seed, double density) {
    Random r = new Random();
    r.setSeed(seed);
    int[][] problem = new int[size][size];
    for (int i = 0; i < problem.length; i++) {
      for (int j = i + 1; j < problem.length; j++) {
        problem[i][j] = r.nextInt(10);
        if (r.nextDouble() > density)
        problem[i][j] = -1;
      }
    }
    return problem;
  }

  private static int[][] fixTwoWayEdges(int[][] problem) {
    for (int i = 0; i < problem.length; i++) {
      problem[i][i] = 0;
```

```
        for (int j = i; j < problem.length; j++) {
          problem[j][i] = problem[i][j];
        }
      }
      return problem;
    }

    public void solve() {
      boolean[] visited = new boolean[problem.length];
      int result = tspsolver(0, visited, 0, "0");

      System.out.println("result: " + result);
      System.out.println("cutoff: " + cutoff);
    }

    /* PRP_PROC */
    public int tspsolver(int node, boolean[] visited, int pathlength, String path) {

      if (pathlength >= cutoff) {
        return -1;
      }
      visited[node] = true;
      boolean wentForward = false;
      int shortestRes = Integer.MAX_VALUE;
      for (int i = 0; i < problem.length; i++) {
        if (!visited[i] && problem[node][i] != -1) {
          wentForward = true;
          /* PRP_CALL */
          int tempres = tspsolver(i, visited.clone(), pathlength
          + problem[node][i], path + " " + i);
          if (tempres < shortestRes && tempres != -1) {
            shortestRes = tempres;
          }
        }
      }
      if (wentForward)
      return shortestRes;
      else {
        // maybe complete path is found
        boolean complete = true;
        for (boolean v : visited) {
          if (!v)
          complete = false;
        }

        if (complete) {
          int thisResult = pathlength + problem[node][0];
          if (thisResult < cutoff) {

            cutoff = thisResult;

          }
          return thisResult;
        } else
```

```
135             return -1;
            }

        }

140 }
```

B.2 Parallell Traveling Salesperson

Listing B.2: PrpTSP.java

```
1  package prpprojects;
   import java.util.Iterator;
   import java.util.concurrent.ExecutorService;
   import java.util.concurrent.ThreadPoolExecutor;
5  import java.util.concurrent.Executors;
   import java.util.concurrent.TimeUnit;
   import java.util.concurrent.CyclicBarrier;
   import java.util.concurrent.BrokenBarrierException;
   import java.util.concurrent.LinkedBlockingQueue;
10 import java.util.ArrayList;
   import java.util.LinkedList;
   import java.util.Stack;
   import java.util.Random;

15 public class PrpTSP {

     public int[][] problem; // problem[i][j] is distance from i to j and

     // from j to i
20
     public volatile int cutoff;

     public PrpTSP(int[][] problem) {
       this.problem = problem;
25     this.cutoff = Integer.MAX_VALUE;

     }

     /**
30   * creates a random problem, and initializes PrpTSP
     *
     * @param args
     */
     public static void main(String[] args) {
35     long start = System.currentTimeMillis();
       if (args.length < 1) {
         System.out
         .println("Usage: java PrpTSP <size> (<random seed>) (<edge density>)");
         System.exit(0);
40     }
       int size = Integer.parseInt(args[0]);

       int seed = 1;
       if (args.length >= 2) {
45       seed = Integer.parseInt(args[1]);
       }
       double density = 1;
       if (args.length >= 3) {
         density = Double.parseDouble(args[2]);
50     }
```

```
      int[][] problem = randomProblem(size, seed, density);
      problem = fixTwoWayEdges(problem);

      // print problem
      System.out.println("problem:");
      for (int i = 0; i < problem.length; i++) {
        for (int j = 0; j < problem.length; j++) {
          System.out.print(problem[i][j] + " ");
        }
        System.out.print("\n");
      }
      PrpTSP solver;

      solver = new PrpTSP(problem);

      solver.solve();

      long end = System.currentTimeMillis();
      long time = end - start;
      System.out.println("Total time " + time + " ms ");
    }

    private static int[][] randomProblem(int size, long seed, double density) {
      Random r = new Random();
      r.setSeed(seed);
      int[][] problem = new int[size][size];
      for (int i = 0; i < problem.length; i++) {
        for (int j = i + 1; j < problem.length; j++) {
          problem[i][j] = r.nextInt(10);
          if (r.nextDouble() > density)
          problem[i][j] = -1;
        }
      }
      return problem;
    }

    private static int[][] fixTwoWayEdges(int[][] problem) {
      for (int i = 0; i < problem.length; i++) {
        problem[i][i] = 0;
        for (int j = i; j < problem.length; j++) {
          problem[j][i] = problem[i][j];
        }
      }
      return problem;
    }

    public void solve() {
      boolean[] visited = new boolean[problem.length];
      int result = tspsolver(0, visited, 0, "0");

      System.out.println("result: " + result);
      System.out.println("cutoff: " + cutoff);
    }
```

```
105
      /* PRP_PROC */
      public int tspsolver(int node, boolean[] visited, int pathlength,
      String path){
        /* admin code */
110     int nCPUs = Runtime.getRuntime().availableProcessors();
        System.out.println("number of CPUs: " + nCPUs);
        int nThreads = nCPUs * 10;
        nWorkers = 0;
        // create stack and FIFO-list
115     Stack<PRPWorker> stack = new Stack<PRPWorker>();
        LinkedList<PRPWorker> fifo = new LinkedList<PRPWorker>();

        // add first worker to the list
        fifo.add(new PRPWorker(node, visited, pathlength, path, ++nWorkers));
120
        // do the following until there are enough workers on the list
        while (fifo.size() < nThreads) {

          // now pop first from the list, and push it on the stackn
125       PRPWorker aWorker = fifo.remove();
          stack.push(aWorker);

          // get it's childrend, and add them to the list
          LinkedList<PRPWorker> children = aWorker.createChildren();
130       fifo.addAll(children);
        }

        // Execute all
        ThreadPoolExecutor executor = new ThreadPoolExecutor(nCPUs, nCPUs,
135     0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());

        for(PRPWorker worker: fifo) {
          executor.execute(worker);
140     }

        // print status header
        System.out.println("\nTime (s) Waiting  Working  Done     Calls/s");

145     try {
          executor.shutdown();

          int time = 0; // second

150       while (!executor.isTerminated()) {
            // wait 1 secound
            synchronized (this) {
              executor.awaitTermination(1, TimeUnit.SECONDS);
            }
155         time++;

            int totalTasks = 0;
            // print some status info
```

```
            for (PRPWorker worker : fifo) {
              if (worker.status > 0) {
                totalTasks += worker.status - worker.oldstatus;
                worker.oldstatus = worker.status;
              }
            }
            printFormatted(time);
            printFormatted(executor.getQueue().size());
            printFormatted(executor.getActiveCount());
            printFormatted((int) executor.getCompletedTaskCount());
            printFormatted(totalTasks);

            System.out.println();
          }

        } catch (InterruptedException e1) {
          e1.printStackTrace();
        }
        // final step. Gather results
        PRPWorker toFinish = null;
        while (!stack.empty()) {
          toFinish = stack.pop();
          toFinish.gatherResults();
        }
        if (toFinish != null) {
          System.out.println("resultatet er " + toFinish.getResult());
          return toFinish.getResult();
        } else
        return 0;
      }

      /**
      * prints the integer using 9 letters, filling out blanks
      */
      private void printFormatted(int i) {
        int colWidth = 8;
        String s = String.valueOf(i);
        System.out.print(s);
        for (int space = s.length(); space < colWidth; space++) {
          System.out.print(" ");
        }
        System.out.print(" ");
      }
      int nWorkers;

      private class PRPWorker implements Runnable {
        int id;
        int status;
        int oldstatus;
        int node;
        boolean[] visited;
        int pathlength;
        String path;
        int result;
```

```
    LinkedList<PRPWorker> childrenList;

215 public PRPWorker(int node, boolean[] visited, int pathlength, String path, int id) {
      this.node = node;
      this.visited = visited;
      this.pathlength = pathlength;
      this.path = path;
220   this.id = id;
      status = 0;
      oldstatus = 0;
      this.childrenList = new LinkedList<PRPWorker>();
    }
225
    public void run() {
      result = solve(node, visited, pathlength, path);
      status = -1;
    }
230
    public LinkedList<PRPWorker> createChildren() {
      childrenList = new LinkedList<PRPWorker>();

      if (pathlength >= cutoff) {
235     ;

      }
      visited[node] = true;
      boolean wentForward = false;
240   int shortestRes = Integer.MAX_VALUE;
      for (int i = 0; i < problem.length; i++) {
        if (!visited[i] && problem[node][i] != -1) {
          wentForward = true;
          /* PRP_CALL */
245       PRPWorker s = new PRPWorker(i, visited.clone(), pathlength
          + problem[node][i], path + " " + i, ++nWorkers);
          childrenList.add(s);
        }
      }
250   return childrenList;
    }
    private int solve(int node, boolean[] visited, int pathlength, String path) {
      status++;

255
      if (pathlength >= cutoff) {
        return -1;
      }
      visited[node] = true;
260   boolean wentForward = false;
      int shortestRes = Integer.MAX_VALUE;
      for (int i = 0; i < problem.length; i++) {
        if (!visited[i] && problem[node][i] != -1) {
          wentForward = true;
265       /* PRP_CALL */
          int tempres = solve(i, visited.clone(), pathlength
```

```
            + problem[node][i], path + " " + i);

            if (tempres < shortestRes && tempres != -1) {
              shortestRes = tempres;
            }
          }
        }
        if (wentForward)
        return shortestRes;
        else {
          // maybe complete path is found
          boolean complete = true;
          for (boolean v : visited) {
            if (!v)
            complete = false;
          }

          if (complete) {
            int thisResult = pathlength + problem[node][0];
            if (thisResult < cutoff) {

              cutoff = thisResult;

            }
            return thisResult;
          } else
          return -1;
        }

      }
      public int gatherResults() {

        if (pathlength >= cutoff) {
          return result =  -1;
        }
        visited[node] = true;
        boolean wentForward = false;
        int shortestRes = Integer.MAX_VALUE;
        for (int i = 0; i < problem.length; i++) {
          if (!visited[i] && problem[node][i] != -1) {
            wentForward = true;
            /* PRP_CALL */
            int tempres = childrenList.remove().getResult();

            if (tempres < shortestRes && tempres != -1) {
              shortestRes = tempres;
            }
          }
        }
        if (wentForward)
        return result =  shortestRes;
        else {
          // maybe complete path is found
          boolean complete = true;
```

```
            for (boolean v : visited) {
              if (!v)
              complete = false;
            }

            if (complete) {
              int thisResult = pathlength + problem[node][0];
              if (thisResult < cutoff) {

                cutoff = thisResult;

              }
              return result =  thisResult;
            } else
            return result =  -1;
          }

        }
        public int getResult() {
          return result;
        }
      }
    }
```

B.3 Goldbach

Listing B.3: Goldbach.java

```
1  package prpprojects;

   import java.io.BufferedWriter;
   import java.io.FileWriter;
5  import java.io.IOException;

   public class GoldBach {

     int[] primes;//array of prime numbers
10
     boolean[] isPrime;

     int limit;

15   int[] gPartitions;

     BufferedWriter out;

     public GoldBach(int limit) {
20     this.limit = limit;
       primes = generatePrimes(limit);
       isPrime = new boolean[limit];
       gPartitions = new int[limit / 2 + 1];
       for (int prime : primes)
25     isPrime[prime] = true;
       try {
         out = new BufferedWriter(new FileWriter("output.dat"));
         out.write("Number of primesums found for each even:\n");
       } catch (IOException e) {
30       // TODO Auto-generated catch block
         e.printStackTrace();
       }
     }

35   /**
     * @param args
     */
     public static void main(String[] args) {

40     long start = System.currentTimeMillis();
       if (args.length < 1) {
         System.out.println("Usage: java Goldbach <limit>");
         System.exit(0);
       }
45     GoldBach gb = new GoldBach(Integer.parseInt(args[0]));

       gb.goldbach();

       long end = System.currentTimeMillis();
50     long time = end - start;
```

```
    System.out.println("Total time " + time + " ms ");
  }

  void goldbach() {

    solve();
    try {
      for (int i = 6; i <= limit; i += 2) {
        out.write(i + ": " + gPartitions[i / 2] + "\n");
      }
      out.close();
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }

  /* PRP_LOOP */
  private void solve() {
    for (int even = 4; even <= limit; even += 2) {
      /* PRP_CALL */
      checkEven(even);

    }
  }

  int checkEven(int even) {
    int i = 0;
    int prime = primes[i];
    while (prime <= even / 2) {
      if (isPrime[even - prime])
      gPartitions[even / 2]++;
      i++;
      prime = primes[i];
    }

    return gPartitions[even / 2];
  }

  /**
  * Generates all primes smaller than input
  *
  * @param n
  * @return an array with all primes between 2 and n
  */
  int[] generatePrimes(int n) {
    boolean a[] = new boolean[n];
    int[] freshPrimes = new int[n];
    int nPrimes = 0;
    a[0] = true;
    a[1] = true;
    int p = 2;
    while (p < n) {
      freshPrimes[nPrimes] = p;
```

```
105           nPrimes++;
              a[p] = true;
              // crossing off all multiples
              for (int i = p * p; i < n && i >= 0; i += p) {
                a[i] = true;
110           }

              // go to next prime
              while (p < n && a[p] == true)
              p++;
115         }

            // trim return array
            int[] allPrimes = new int[nPrimes];
            for (int i = 0; i < nPrimes; i++) {
120           allPrimes[i] = freshPrimes[i];
            }
            return allPrimes;
        }
    }
```

B.4 Parallell Goldbach

Listing B.4: PrpGoldbach.java

```
1  package prpprojects;

   import java.util.Iterator;
   import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.ThreadPoolExecutor;
   import java.util.concurrent.Executors;
   import java.util.concurrent.TimeUnit;
   import java.util.concurrent.CyclicBarrier;
   import java.util.concurrent.BrokenBarrierException;
10 import java.util.concurrent.LinkedBlockingQueue;
   import java.util.ArrayList;
   import java.util.LinkedList;
   import java.util.Stack;
   import java.io.BufferedWriter;
15 import java.io.FileWriter;
   import java.io.IOException;

   public class PrpGoldBach {
     int[] primes;
20
     boolean[] isPrime;

     int limit;

25   int[] gPartitions;

     BufferedWriter out;

     public PrpGoldBach(int limit) {
30     this.limit = limit;
       primes = generatePrimes(limit);
       isPrime = new boolean[limit];
       gPartitions = new int[limit / 2 + 1];
       for (int prime : primes)
35     isPrime[prime] = true;
       try {
         out = new BufferedWriter(new FileWriter("output.dat"));
         out.write("Number of primesums found for each even:\n");
       } catch (IOException e) {
40       // TODO Auto-generated catch block
         e.printStackTrace();
       }
     }

45   /**
     * @param args
     */
     public static void main(String[] args) {

50     long start = System.currentTimeMillis();
```

```
    if (args.length < 1) {
      System.out.println("Usage: java Goldbach <limit>");
      System.exit(0);
    }
    PrpGoldBach gb = new PrpGoldBach(Integer.parseInt(args[0]));

    gb.goldbach();

    long end = System.currentTimeMillis();
    long time = end - start;
    System.out.println("Total time " + time + " ms ");
  }

  void goldbach() {

    solve();
    try {
      for (int i = 6; i <= limit; i += 2) {
        out.write(i + ": " + gPartitions[i / 2] + "\n");
      }
      out.close();
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }

  /* PRP_LOOP */
  private void solve() {
    int nCPUs = Runtime.getRuntime().availableProcessors();
    // System.out.println("number of CPUs: " + nCPUs);
    // System.out.println("level " + level);
    int nThreads = nCPUs;
    barrier = new CyclicBarrier(nThreads);

    // store the LoopWorkers in an ArrayList instead of an array in case of
    // generic class. Array of generic class does not work
    ArrayList<LoopWorker> workers = new ArrayList<LoopWorker>(nThreads);
    ExecutorService executor = Executors.newFixedThreadPool(nCPUs);

    for (int cpu = 0; cpu < nThreads; cpu++) {
      LoopWorker l = new LoopWorker(cpu, nThreads);
      executor.execute(l);
      workers.add(l);
    }

    // print status header
    final int colWidth = 8;
    System.out.println("\nNumber of tasks solved each second:");
    System.out.print("Time (s) ");
    for (LoopWorker lw : workers) {
      String header = "Thr " + lw.id + "    ";
      System.out.print(header.substring(0, colWidth));
    }
```

```
105       System.out.println();

          try {
            executor.shutdown();
            int time = 0;
110         while (!executor.isTerminated()) {
              // wait 1 second
              synchronized (this) {
                executor.awaitTermination(1, TimeUnit.SECONDS);
              }
115           time++;
              String timeString = String.valueOf(time);
              System.out.print(time);
              for (int space = timeString.length(); space < 9; space++)
              System.out.print("␣");
120           // print some status info
              for (LoopWorker lw : workers) {
                if (lw.status > 0) {
                  int tasks = lw.status - lw.oldstatus;
                  lw.oldstatus = lw.status;
125               String taskstring = String.valueOf(tasks);
                  System.out.print(taskstring);
                  for (int space = taskstring.length(); space < colWidth; space++)
                  System.out.print("␣");
                } else if (lw.status == 0) {
130               System.out.print("waiting␣");
                } else if (lw.status < 0) {
                  System.out.print("done␣␣␣␣");
                }

135           }
              System.out.println();
            }
            // executor.awaitTermination(10000000, TimeUnit.SECONDS);
          } catch (InterruptedException e1) {
140         e1.printStackTrace();
          }
        }

        CyclicBarrier barrier;
145
        private class LoopWorker implements Runnable {
          int id;

          int nWorkers;
150
          int currentWorker;

          int status;

155       int oldstatus;

          public LoopWorker(int id, int nWorkers) {
            this.id = id;
```

```
        this.nWorkers = nWorkers;
        this.currentWorker = 0;
        this.status = 0;
        this.oldstatus = 0;
      }

      public void run() {

        for (int even = 4; even <= limit; even += 2) {
          /* PRP_CALL */
          currentWorker++;
          if (currentWorker % nWorkers == id) {
          checkEven(even);
          status++;
        }

      }

      status = -1;
    }
  }

  int checkEven(int even) {
    int i = 0;
    int prime = primes[i];
    while (prime <= even / 2) {
      if (isPrime[even - prime])
      gPartitions[even / 2]++;
      i++;
      prime = primes[i];
    }

    return gPartitions[even / 2];
  }

  /**
  * Generates all primes smaller than input
  *
  * @param n
  * @return an array with all primes between 2 and n
  */
  int[] generatePrimes(int n) {
    boolean a[] = new boolean[n];
    int[] freshPrimes = new int[n];
    int nPrimes = 0;
    a[0] = true;
    a[1] = true;
    int p = 2;
    while (p < n) {
      freshPrimes[nPrimes] = p;
      nPrimes++;
      a[p] = true;
      // crossing off all multiples
      for (int i = p * p; i < n && i >= 0; i += p) {
```

```
        a[i] = true;
      }

      // go to next prime
      while (p < n && a[p] == true)
      p++;
    }

    // trim return array
    int[] allPrimes = new int[nPrimes];
    for (int i = 0; i < nPrimes; i++) {
      allPrimes[i] = freshPrimes[i];
    }
    return allPrimes;
  }
  }
```

B.5 Optimalt Søketre

Listing B.5: OstBuilder.java

```
1  package prpprojects;

   public class OstBuilder<V extends Comparable<V>> {

5    final int n;

     final V[] values;

     final int[] p;
10
     final int[][] a;

     final int[][] root;

15   public long time;

     public void buildTree() {
       long start = System.currentTimeMillis();

20     for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
           a[i][j] = -1;
           // l[i][j] = new ReentrantLock();
           root[i][j] = -1;
25       }
       }

       // fill first level
       for (int i = 0; i < n; i++) {
30       a[i][i] = p[i];
         root[i][i] = i;
       }
       // fill sigma
       for (int i = 1; i < n; i++) {
35       for (int j = 0; j < i; j++) {
           a[i][j] = a[i - 1][j] + p[i];
         }
       }

40     pump();

       System.out.println("\ncost: " + a[0][n - 1]);
       long end = System.currentTimeMillis();
       time = end - start;
45     System.out.println("Total time " + time + " ms ");

     }

     /* PUMP_LOOP */
50   private void pump() {
```

```
      //int complete = -1;

      for (int level = 1; level < n; level++) {

        //PUMP_CALL
        solveLevel(level);
        // solve my tasks on this level. Store locally

        /*
        int oldComplete = complete;

        complete = (int) (100 * (2 + level) / n);
        if (complete != oldComplete) {
          if (oldComplete < 10 && oldComplete >= 0)
          System.out.print("\b\b\b");
          else if (oldComplete >= 0)
          System.out.print("\b\b\b\b");

          System.out.print(complete + " %");
        }
        */
      }

    }

    /* PRP_LOOP */
    private void solveLevel(int level) {
      int i = 0;

      for (int j = level; j < n; j++) {

        /* PRP_CALL */
        solve(i, j);
        i++;
      }
    }

    /**
    * Solves a[i][j] by reading parts from a[][] and storing result in locala.
    * also writes a number in root.
    *
    * ps: no writing til a!, and no reading from root
    *
    *
    * @param i
    * @param j
    */
    private void solve(int i, int j) {

      int resultat = Integer.MAX_VALUE;
      for (int k = i; k <= j; k++) {

        int sum = 0;
        if (k > i)
```

```
105           sum += a[i][k - 1];
              if (k < j)
              sum += a[k + 1][j];

              if (sum < resultat) {
110             resultat = sum;
                // root[i][j] = k;
              }

            }
115
            a[i][j] = resultat + a[j][i];// +=sigma
          }

          public static void main(String[] args) {
120         OstBuilder<Integer> builder;

            if (args.length == 0) {
              System.out.println("usage: java OstBuilder <size>");
            }
125         if (args.length == 1) {

              int size = Integer.parseInt(args[0]);
              int[] problem = OstProblems.generate(size);
              Integer[] values = new Integer[size];
130           for (int i = 0; i < size; i++) {
                values[i] = i;
              }
              builder = new OstBuilder<Integer>(values, problem);

135           builder.buildTree();
            }

          }

140       public OstBuilder(V[] values, int[] p) {
            this.n = values.length;
            this.values = values;
            this.p = p;

145         a = new int[n][n];
            root = new int[n][n];
          }

          public int getCost() {
150         return a[0][n - 1];
          }

          public long getTime() {
            return time;
155       }
        }
```

B.6 Parallell Optimalt Søketre

Listing B.6: PrpOstBuilderjava

```
1  package prpprojects;

   import java.util.Iterator;
   import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.ThreadPoolExecutor;
   import java.util.concurrent.Executors;
   import java.util.concurrent.TimeUnit;
   import java.util.concurrent.CyclicBarrier;
   import java.util.concurrent.BrokenBarrierException;
10 import java.util.concurrent.LinkedBlockingQueue;
   import java.util.ArrayList;
   import java.util.LinkedList;
   import java.util.Stack;

15 public class PrpOstBuilder<V extends Comparable<V>> {

     final int n;

     final V[] values;
20
     final int[] p;

     final int[][] a;

25   final int[][] root;

     public long time;

     public void buildTree() {
30     long start = System.currentTimeMillis();

       for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
           a[i][j] = -1;
35         root[i][j] = -1;
         }
       }

       // fill first level
40     for (int i = 0; i < n; i++) {
         a[i][i] = p[i];
         root[i][i] = i;
       }
       // fill sigma
45     for (int i = 1; i < n; i++) {
         for (int j = 0; j < i; j++) {
           a[i][j] = a[i - 1][j] + p[i];
         }
       }
50
```

```
    pump();

    System.out.println("\ncost: " + a[0][n - 1]);
    long end = System.currentTimeMillis();
    time = end - start;
    System.out.println("Total time " + time + " ms ");

  }

  /* PUMP_LOOP */
  private void pump() {
    int nCPUs = Runtime.getRuntime().availableProcessors();
    int nThreads = nCPUs;
    barrier = new CyclicBarrier(nThreads);

    // store the LoopWorkers in an ArrayList instead of an array in case of
    // generic class. Array of generic class does not work
    ArrayList<LoopWorker> workers = new ArrayList<LoopWorker>(nThreads);
    ExecutorService executor = Executors.newFixedThreadPool(nCPUs);

    for (int cpu = 0; cpu < nThreads; cpu++) {
      LoopWorker l = new LoopWorker(cpu, nThreads);
      executor.execute(l);
      workers.add(l);
    }

    // print status header
    final int colWidth = 8;
    System.out.println("\nNumber of tasks solved each second:");
    System.out.print("Time (s) ");
    for (LoopWorker lw : workers) {
      String header = "Thr " + lw.id + "     ";
      System.out.print(header.substring(0, colWidth));
    }
    System.out.println("Pump");

    try {
      executor.shutdown();
      int time = 0;
      while (!executor.isTerminated()) {
        // wait 1 second
        synchronized (this) {
          executor.awaitTermination(1, TimeUnit.SECONDS);
        }
        time++;
        String timeString = String.valueOf(time);
        System.out.print(time);
        for (int space = timeString.length(); space < 9; space++)
        System.out.print(" ");
        // print some status info
        for (LoopWorker lw : workers) {
          if (lw.status > 0) {
            int tasks = lw.status - lw.oldstatus;
            lw.oldstatus = lw.status;
```

```
105               String taskstring = String.valueOf(tasks);
                  System.out.print(taskstring);
                  for (int space = taskstring.length(); space < colWidth; space++)
                  System.out.print(" ");
                } else if (lw.status == 0) {
110               System.out.print("waiting ");
                } else if (lw.status < 0) {
                  System.out.print("done    ");
                }

115           }
              System.out.print(workers.get(0).pumpCall);
              System.out.println();
            }
            // executor.awaitTermination(10000000, TimeUnit.SECONDS);
120       } catch (InterruptedException e1) {
            e1.printStackTrace();
          }
        }

125     CyclicBarrier barrier;

        /* PRP_LOOP */
        private class LoopWorker implements Runnable {
          int id;
130
          int nWorkers;

          int currentWorker;

135       int status;

          int oldstatus;

          int pumpCall;
140
          public LoopWorker(int id, int nWorkers) {
            this.id = id;
            this.nWorkers = nWorkers;
            this.currentWorker = 0;
145         this.status = 0;
            this.oldstatus = 0;
            this.pumpCall = 0;
          }

150       public void run() {

            // int complete = -1;

            for (int level = 1; level < n; level++) {
155
              // PUMP_CALL
              pumpCall++;
              solveLevel(level);
```

```
            try {
              barrier.await();
            } catch (InterruptedException e) {
              e.printStackTrace();
            } catch (BrokenBarrierException e) {
              e.printStackTrace();
            }

            // solve my tasks on this level. Store locally

            /*
            * int oldComplete = complete;
            *
            * complete = (int) (100 * (2 + level) / n); if (complete !=
            * oldComplete) { if (oldComplete < 10 && oldComplete >= 0)
              * System.out.print("\b\b\b"); else if (oldComplete >= 0)
              * System.out.print("\b\b\b\b");
              *
              * System.out.print(complete + " %"); }
              */
            }

            status = -1;
          }

          private void solveLevel(int level) {
            int i = 0;

            for (int j = level; j < n; j++) {

              /* PRP_CALL */
              currentWorker++;
              if (currentWorker % nWorkers == id) {
              solve(i, j);
              status++;
            }

            i++;
          }
        }
      }

      /**
      * Solves a[i][j] by reading parts from a[][] and storing result in locala.
      * also writes a number in root.
      *
      * ps: no writing til a!, and no reading from root
      *
      *
      * @param i
      * @param j
      */
      private void solve(int i, int j) {
        int resultat = Integer.MAX_VALUE;
```

```
      for (int k = i; k <= j; k++) {
        int sum = 0;
        if (k > i)
        sum += a[i][k - 1];
        if (k < j)
        sum += a[k + 1][j];

        if (sum < resultat) {
          resultat = sum;
          // root[i][j] = k;
        }

      }

      a[i][j] = resultat + a[j][i];// +=sigma
    }

    public static void main(String[] args) {
      PrpOstBuilder<Integer> builder;

      if (args.length == 0) {
        System.out.println("usage: java PrpOstBuilder <size>");
      }
      if (args.length == 1) {

        int size = Integer.parseInt(args[0]);
        int[] problem = OstProblems.generate(size);
        Integer[] values = new Integer[size];
        for (int i = 0; i < size; i++) {
          values[i] = i;
        }
        builder = new PrpOstBuilder<Integer>(values, problem);

        builder.buildTree();
      }

    }

    public PrpOstBuilder(V[] values, int[] p) {
      this.n = values.length;
      this.values = values;
      this.p = p;

      a = new int[n][n];
      root = new int[n][n];
    }

    public int getCost() {
      return a[0][n - 1];
    }

    public long getTime() {
      return time;
    }
```

```
}
```

Bibliografi

[BOI08] BOINCstats. Seti@home statistikk. Online, 2008. http://boincstats.com/stats/project_graph.php?pr=sah.

[Dev08] Advanced Micro Devices. Amd phenom news. Online, 2008. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543 124395,00.html.

[Dig08] Western Digital. Western digital raptor specifications. Online, 2008. http://www.westerndigital.com/en/products/Products.asp?DriveID=189#jump11.

[Eid98] Victor Eide. Parallel recursive procedures, a manager/worker approach. Master's thesis, UiO, 1998.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtous organizations. *International Journal of High Performance Computing Applications*, 2001.

[Fos95] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1.3 edition, 1995.

[Gar08] Gartner. Gartner identifies seven grand challenges facing it. Online, 2008. .

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Sun Microsystems, second edition, 2000.

[Goe01] Brian Goetz. Threading lightly, part 1: Synchronization is not the enemy. Online, 2001. http://www.ibm.com/developerworks/java/library/j-threads1.html.

[IBM08] IBM. Ibm 3340 direct access storage facility. Online, 2008. http://www-03.ibm.com/ibm/history/exhibits/storage/storage_3340.html.

[Int07] Intel. Intel demonstrates industry's first 32nm chip and next-generation nehalem microprocessor architecture. Online, 2007. http://www.intel.com/pressroom/archive/releases/20070918corp_a.htm.

[Kan03] Michael Kanellos. Intel scientists find wall for moore's law. Online, 2003. .

[MA95] Arne Maus and Torfinn Aas. Prp - parallel recursive procedures. *PRP*, 1995.

[Mat08] Wolfram MathWorld. Goldbach conjecture. Online, 2008. http://mathworld.wolfram.com/GoldbachConjecture.html.

[Mic04] Sun Microsystems. Package java.util.concurrent.atomic. Online, 2004. http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrentatomic/package-summary.html.

[Mic08] Sun Microsystems. Atomic operations. Online, 2008. http://java.sun.com/docs/books/tutorial/essential/concurrency/atomic.html.

[Moo05] Gordon Moore. Excerpts from a conversation with gordon moore. *Intel Corporation*, 2005.

[MV99] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: Problems and solutions. *ACM Crossroads*, 1999. http://www.acm.org/crossroads/xrds5-3/pmgap.html.

[Ope08] OpenMP. About openmp. Online, 2008. http://openmp.org/wp/about-openmp/.

[PCW07] PCWorld. Intel chip news. Online, 2007. http://www.pcworld.com/article/id,128924-c,intel/article.html.

[Syv07] Jørn Christian Syversen. Forbedring av kjøresystemet i prp. Master's thesis, UiO, 2007.

[Tan01] Andrew Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2 edition, 2001.

[TOP08] TOP500.org. Top500 supercomputing sites. Online, 2008. http://www.top500.org.

[Uni06] Rice University. High performance fortran. Online, 2006. http://hpff.rice.edu/.

[Wei99a] Mark Allen Weiss. Binary trees. In *Data Structures & Algorithm Analysis In Java*, chapter 4.2. Addison Wesley Longman, 1999.

[Wei99b] Mark Allen Weiss. A brief introduction to recursion. In *Data Structures & Algorithm Analysis In Java*, chapter 1.3. Addison Wesley Longman, 1999.

[Wei99c] Mark Allen Weiss. Dynamic programming. In *Data Structures & Algorithm Analysis In Java*, chapter 10.3. Addison Wesley Longman, 1999.

[Wei99d] Mark Allen Weiss. Introduction to np-completeness. In *Data Structures & Algorithm Analysis In Java*, chapter 9.7. Addison Wesley Longman, 1999.

[Wei99e] Mark Allen Weiss. Optimal binary search tree. In *Data Structures & Algorithm Analysis In Java*, chapter 10.3.3. Addison Wesley Longman, 1999.

[Wik08a] Wikipedia. Hyper-threading. Online, 2008. http://en.wikipedia.org/wiki/Hyper_threading.

[Wik08b] Wikipedia. List of intel microprocessors. Online, 2008. http://en.wikipedia.org/wiki/List_of_Intel_microprocessors.

Figurer

Tabeller

Listings